2018-12-06

Extension on Adaptive MAC Protocol for Space Communications

Max Hongming Li
Worcester Polytechnic Institute, mhli@wpi.edu

Extension on Adaptive MAC Protocol for Space Communications

by

Max Li

A Master's Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Electrical and Computer Engineering

Dec 2018

APPROVED:

Professor Alex Wyglinski, WPI, Advisor

Professsor D. Richard Brown III, WPI

Professor Sven G. Bilén, The Pennsylvania State University

Abstract

This work devises a novel approach for mitigating the effects of Catastrophic Forgetting in Deep Reinforcement Learning-based cognitive radio engine implementations employed in space communication applications. Previous implementations of cognitive radio space communication systems utilized a moving window-based online learning method, which discards part of its understanding of the environment each time the window is moved. This act of discarding is called Catastrophic Forgetting. This work investigated ways to control the forgetting process in a more systematic manner, both through a recursive training technique that implements forgetting in a more controlled manner and an ensemble learning technique where each member of the ensemble represents the engine's understanding over a certain period of time. Both of these techniques were integrated into a cognitive radio engine proof-of-concept, and were delivered to the SDR platform on the International Space Station. The results were then compared to the results from the original proof-of-concept. Through comparison, the ensemble learning technique showed promise when comparing performance between training techniques during different communication channel contexts.

Acknowledgements

I would first like to thank Professor Alex M. Wyglinski for advising me throughout the entire process of this thesis, from providing me with the opportunity to work on this project to providing support and critique during the writing of this document. Without his guidance I would likely have had a much more circuitous path to completion of the thesis.

This research was supported by contributions provided by NASA's John H. Glenn Research Center, as part of an extension of previous work. Special thanks to Richard Reinhart and Dale Mortensen for the expertise provided and administrative assistance provided, as well the rest of the SCaN testbed team who ensured that the testing was conducted as smoothly as possible.

Many thanks to Timothy Hackett from Penn State, the student who worked on the Cognitive Engine prior to me. He a major resource on the existing Cognitive Engine code base, as well as how to run the ground and flight tests. His assistance on the first days of ground testing and flight testing was very helpful.

Finally, a thanks to Professor Sven G. Bilén of the Pennsylvania State University and Professor D. Richard Brown of WPI for being on my M.S. committee and providing feedback on this document.

Contents

List of Figures

List of Tables

Nomenclature

E_s/N_0 Energy per symbol to noise power spectral density (symbol-based SNR measurement)

ACM Adaptive Coding and Modulation

AI Artificial Intelligence

ALI Adversarial Learning Interface

API Application Program Interface

ARCC Advanced Radio for Cognitive Communications

BER Bit Error Rate

CART Classification and Regression Tree

CE Cognitive Engine

CE-LM Cognitive Engine - Levenberg-Marquardt training

CE-NSE Cognitive Engine - Learn++.NSE training

CE-RLM Cognitive Engine - Recursive Levenberg-Marquardt training

CIFAR-10 Canadian Institute For Advanced Research 10-Class dataset

CNN Convolutional Neural Network

CR Cognitive Radio

DCGAN Deep Convolutional Generative Adversarial Network

DL Deep Learning

DQN Deep-Q Network

DVB-S Digital Video Broadcasting – Satellite

DVB-S2 Digital Video Broadcasting – Satellite – Second Generation

DVB-S2X Digital Video Broadcasting – Satellite – Second Generation Extension

FFNN FeedForward Neural Network

FPGA Field Programmable Gate Array

GAN Generative Adversarial Network

GEO Geosynchronous Equatorial Orbit

GPL General Public License

GPP General Purpose Processor

GPU Graphics Processing Unit

GRC Glenn Research Center

ITAR International Traffic in Arms Regulations

LEO Low Earth Orbit

LM Levenberg-Marquardt

MAC Medium Access Control

ML Machine Learning

MLP Multilayer Perceptron

MNIST Modified National Institute of Standards and Technology Dataset

MPEG-2 Data compression standard, developed to improve on MPEG-1

MPEG-4 Data compression standard, expanding upon the improvements of MPEG-2

MSE Mean Squared Error

NASA National Aeronautics and Space Administration

NSE Nonstationary Environment

OSI Open Systems Interconnection

PHY Physical Access Layer

RL Reinforcement Learning

SARSA State-Action-Reward-State-Action

SCaN Space Communications and Navigation

SDR Software Defined Radio

SNR Signal-to-Noise Ratio

STRS Space Telecommunications Radio System

TD Temporal Difference

TDRSS Tracking and Data Relay Satellite System

TFD Toronto Face Dataset

UDP User Datagram Protocol

VM Virtual Machine

WGAN Wasserstein Generative Adversarial Network

Chapter 1

Introduction

1.1 Motivation

In recent times, the amount of computing power available has increased to the point where many techniques that have previously been considered too complex for certain time-sensitive applications have become feasible [3]. The increase in computing power includes the general purpose processors (GPPs) that most people are familiar with, but also graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), which are more suited to specialized applications. One field with potential to benefit from these developments is space communications, for which an increase in computing power enables the use of Software Defined Radios (SDRs) [4,5]. SDRs provide flexibility by enabling many aspects of communications that are typically implemented in hardware to be implemented in software. This includes tasks such as such as filtering and detection of signals, as well as higher level things like reconfiguration of applications and services. [6, p. xxxiii]. The idea of software implementation of hardware elements is not particularly new, but was previously difficult to implement with the computing power available. As the computing power at any given price point increased, cognitive communication algorithms that utilize

the manuverability of SDRs were proposed. Many of the algorithms that have been implemented and tested focus on employing dynamic spectrum access techniques or other sensing approaches. In more recent times, many technologies have taken a different approach by attempting to achieve multi-objective goals through the modification of multiple radio parameters in a manner that rewards good behaviors. This learning behavior sets a foundation for the development of cognitive systems. This increased focus on multi-objective goals has been in part motivated by the satellite communications industry, which has proposed business ventures involving hundreds of spacecraft. With satellite communications, objectives have the potential to interact in complicated ways and may be inversely related in some cases. Maintaining "good" performance requires a balancing of these objectives. These large networks of satellites, combined with the recent interest in space exploration, motivate communication methods that have high link availability and robustness [7]. A cognitive communications system is likely necessary in acheiving both of these requirements satisfatorily, especially for situations where autonomous operation is required.

One of the more general purpose technologies that has experienced large growth from the boom in processing power is Artificial Intelligence (AI), or more specifically Machine Learning (ML). Breakthroughs like IBM's Watson [8] and DeepMind's AlphaGo [9] have been driven by the ability to use large amounts of computing power to apply increasingly more complex Deep Learning (DL) techniques, such as convolutional neural networks (CNNs) and deep-Q networks (DQNs) [9]. These techniques have typically involved too many computations to be considered tractible for many problems. However, the recent explosion of computing power available has enabled the use of these DL techniques that reach performance and speed thresholds previously thought impossible. This allows for a whole variety of new fields of application that may have stricter timing or accuracy requirements.

1.2 State of the Art

In this section, an overview of recent developments in satellite communications and cognitive radio as they relate to this thesis will be provided.

1.2.1 Satellite Communications

Within the field of satellite communications, the most relevent developments are in the fields of Adaptive Coding and Modulation (ACM) schemes and satellite transmission standards. ACM is used in the situation that received signal power changes due to impairments in the channel [10]. This change in power is often a result of fading, which can be caused by weather conditions or the relative motion between the transmitter and the receiver [11], among other things. ACM adaptively chooses the modulation and coding scheme based on the observed link budget and the quality of the message at the receiver, usually by looking for the optimal configuration in a table. ACM has been applied to Geosynchronous Equatorial Orbit (GEO) satellite channels operating in the S-band [12] as well as the Ka-band [13]. In this work, it is considered to be the standard solution for Low Earth Orbit (LEO) satellite links.

In the context of satellite communications, there are a number of different standards that can be used in transmission of data. For this thesis, DVB-S2 [14] was chosen as the standard that represents most modern satellites. This standard is the second generation of DVB-S, a technical standard for GEO satellite-based digital television brodcast systems that was primarily desigend for direct-to-home services. Most of the innovations of DVB-S2 are present in the physical (PHY) layer in the OSI model [15], with more e-client channel coding, modulation, and error correction techniques. In addition, it uses recent video compression technology to enable transmissions compatable with MPEG-2 and MPEG-4 [16] standards. DVB-S2 also utilizes a powerful forward error correction scheme that allows for four modulation

constellations at a variety of code rates. The selection of modulation constellations and code rates can be controlled by an ACM scheme that allows for modulation on a frame-by-frame basis. In practice, DVB-S2 and its most recent extension DVB-S2X [17] have improved performance for mobile applications by allowing channel bonding, which combines unused portions of spectrum into a single virtual channel that can provide a higher bandwidth.

1.2.2 Cognitive Radio

Within the SDR literature, the concept of on-board cognition has been considered the likely next technological breakthrough. On-board cognition enables environmental awareness across several Open Systems Interconnection (OSI) layers [18], real-time knowledge of channel conditions, and assessment of currently available resources, among other things. This information is a very important aspect of optimizing the communications link performance. In the past, a variety of different algorithms were applied to cognitive radios [19], including genetic algorithms [20]. Genetic algorithms in particular do not always converge, and might take several iterations of the algorithm to find a stable solution. With each change in environmental conditions requiring another set of iterations to find the new stable solution, the time for training and retraining makes genetic algorithms unlikely to be a good fit for the rapidly updating environment of satellite communications, and CR in general.

ML techniques have been studied for use in CR [21,22]. Both studies investigated how optimization and ML can be used in assisting CR systems to find the best configuration parameter set. The majority of research focuses on spectrum management and sensing techniques for terrestrial links [22, 23]. Research on satellite links has similar focus points, with the majority of CR research focusing on spectrum resource allocation [24,25]. At the time of the writing of this thesis, there has been very little

focus on radio resource managementfor point-to-point communication links.

For this thesis, communications performance is evaluated as a holistic combination of pre-existing performance metrics, including minimum Bit Error Rate(BER), maximum throughput, and power adaptation. During critical space mission phases, communications systems may need to manage resources while encountering conflicting performance requirements and limited resource availability.

1.3 Research Contributions

In the research that directly precedes this thesis [1, 2], a Cognitive Engine (CE) architecture for adapting PHY parameters for space communications was developed and tested. The CE combines a DQN with an "Explore" network modelling the relationship between the action space (which in this case is the configurable PHY parameters) and certain evaluation metrics relevant to communications. The DQN is used to choose a high quality action, while the Explore network is used to steer exploration of the environment to the subset of actions that it expects to get good results. By doing this, it is able to learn which actions are appropriate in any given channel condition.

This thesis extends the previous research by investigating methods to mitigate Catastrophic Forgetting, a side effect of constantly retraining the DQN and the Explore network. Multiple different training methods, including a recursive method and an ensemble-based method, were developed and tested. First, a MATLAB simulation was used to verify the validity of recursive and ensemble methods as improvements on the base CE. Then, through C++ simulations and testing with the ISS, Learn++.NSE was determined to be the better way of addressing Catastrophic Forgetting, getting closer to the optimal fitness score than the baseline CE training method.

1.4 Thesis Outline

In Chapter 2, relevant aspects of ML and the details about previous work on the project are described. Following this, details of the modified CE architecture, the MATLAB simulation, and the C++ implementation are described in Chapter 3, along with the hardware setups used for testing. Chapter 4 describes the simulation and flight testing results. Finally, Chapter 5 summarizes the concluding points and describes future work.

Chapter 2

Overview on Machine Learning Applied to Satellite Communications

In this chapter, an overview of the SCaN Testbed Cognitive Engine (CE) is provided. In order to understand and motivate decisions that were made in the initial development, general concepts of machine learning will first be discussed. Then, the work accomplished through NASA's SCaN Testbed project prior to this thesis will be described. After this, topics that are relevant to the improvements introduced to the CE by this thesis will be discussed. Finally, the chapter will be summarized.

2.1 A General Overview of Machine Learning

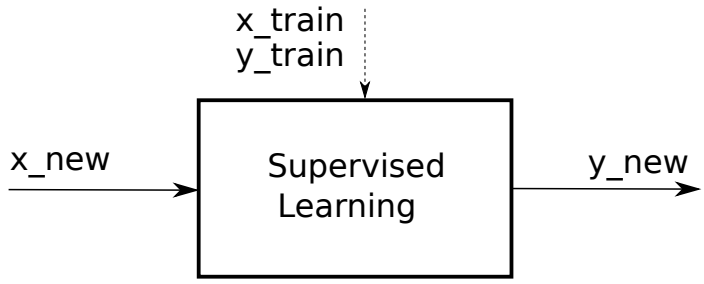
With the advent of increasingly more capable and accessible processors, machine learning has become a useful tool in many different fields [26] [27] [28]. This is in part driven by the flexibility and efficiency that machine learning is capable of. There are three major categories of ML algorithms [29]: supervised learning, unsupervised learning, and reinforcement learning. Supervised learning takes a set of input values

X and a set of target results Y in order to create a mapping $x \rightarrow g(x) \rightarrow y$. Once the algorithm is trained, this mapping $g(x)$ can be used with new input values x_{new} to predict new target values y_{new}. This process is shown in Figure 2.1. Unlike supervised learning, unsupervised learning is only given a set of input values X. With this set of input values, unsupervised learning attempts to understand the inherent structure within the input values [29]. In more general terms, unsupervised learning is attempting to make inferences based on the X given to it. Reinforcement learning is unlike either of the two previously described categories. The primary focus of reinforcement learning is to understand the environment surrounding the learning agent by mapping situations to actions based on maximizing a reward value [30]. Different supervised learning, where there is an X and Y given to find $g(x)$, reinforcement learning algorithms have X and some knowledge of the reward behavior of the environment $R(x)$, and are trying to choose X in order to get a large Y while simultaneously expanding its knowledge of $R(x)$.

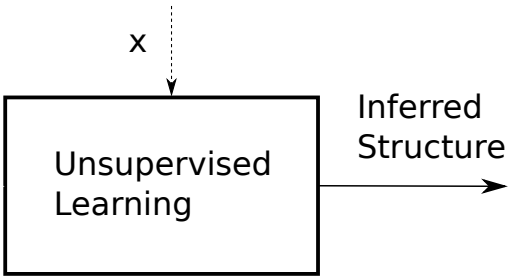
Within the CE, concepts from Supervised Learning and Reinforcement Learning are used. As such, the rest of this section will focus on those two categories in more detail.

2.1.1 Supervised Learning

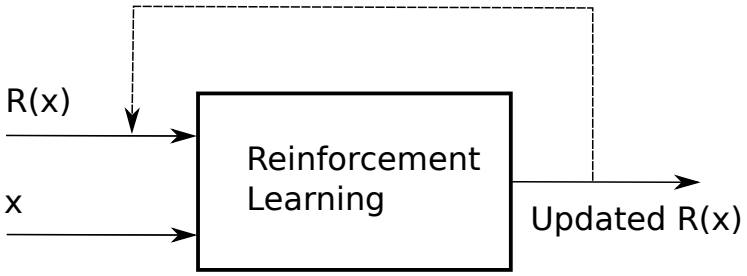
As stated before, supervised learning refers to the problem of modeling some sort of relation between two sets of data: an input dataset and a target dataset. This problem is further subdivided into two different types of problems: regression and classification. Regression refers to the modeling of a continuous target. One common example is the modeling of the cost of a house based on its square footage. x in this case would be the square footage of the house, and y would be the cost of the house. Regression refers to the fact that the cost of a house can have any value within a range. The other type of problem is classification, which models a discrete target.



(a) Supervised Learning Block Diagram.



(b) Unsupervised Learning Block Diagram.



(c) Reinforcement Learning Block Diagram.

Figure 2.1: Flow diagrams of the three main categories of Machine Learning.

This is the act of assigning input values to one of a number of different classes. A simple to understand example is determining if someone is likely to renew a magazine based on their age. In this case x is the age of the subscriber, and y is whether or not the subscriber will renew their subscription. This is classification because there are two discrete possibilities for y, that the subscriber will renew their subscription and that the subscriber will not renew their subscription.

In order to train a model for either problem class, a set of training data similar to the expected input is required, X_{train}. In addition to the input set, the corresponding

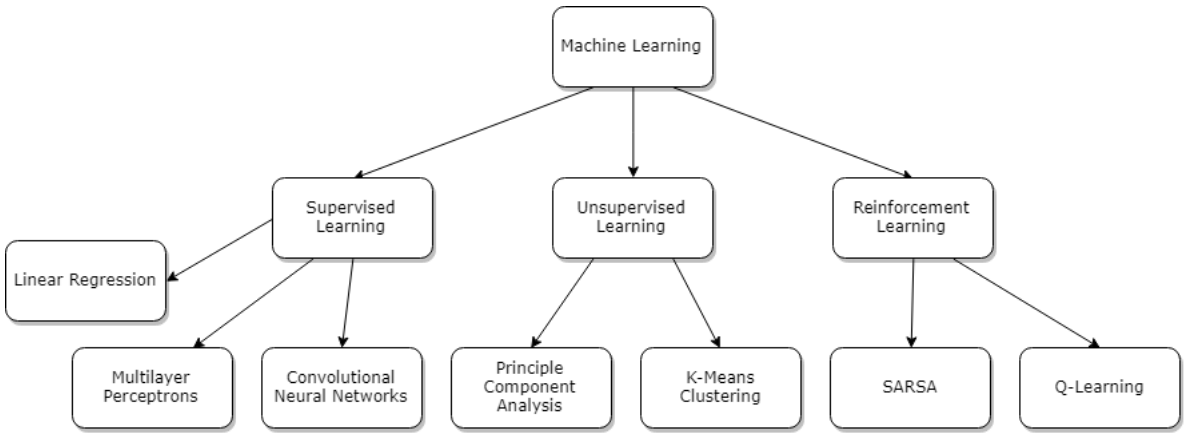


Figure 2.2: Basic overview of different machine learning techniques.

target values are needed as well, Y_{train}. With this information, the chosen algorithm is trained to understand the nonlinear mapping from input to target. Once the algorithm is trained, it will take inputs X and predict the resulting value of Y. There are a wide variety of algorithms that are applied to supervised learning. Some relevant algorithms include Linear Regression, Classification and Regression Trees (CART), Multilayer Perceptrons/Feed Forward Neural Networks (MLP/FFNN), and Convolutional Neural Networks (CNN). Each of these algorithms has its own training method. A focused description of the training algorithms for CART and MLP will be provided. CART was chosen because of the relative ease of explanation and the fact that it is often used as a baseline of performance to compare to. MLP was chosen because it is the method used in CE.

The CART is one of the most basic supervised learning techniques still in use today. For each input vector $\vec{x} \in X$, a series of decisions are made based on the values of $\vec{x}$. At each decision, the possible state of the tree is split in two, depending on which value $\vec{x}$ takes. Once there are no more splits, the tree comes to either a class (for classification) or a value (for regression) for $\vec{x}$. This is more thoroughly described in Algorithm 1.

One benefit of this algorithm is that it is fairly intuitive, behaving in a way that is similar to how a human might make a decision. It is also fairly simple, not making

Algorithm 1 CART Pseudoalgorithm

```
1: procedure CART
2:     while leafNotReached do
3:         x = x_eval ∈ X
4:         if x > branchValue then Branch right.
5:         else Branch left.
6:     return leafValue or leafClass
```

any underlying assumptions about the underlying data or requiring any parameters. This allows CART to be applied in a fairly straighforward manner to a wide variety of problems. One downside of CART is that its simplicity makes the method prone to overfitting. Given n inputs, a tree that splits $n-1$ ways would be capable of 100% accuracy on the training dataset. However, it would likely underperform on any other data, with stochastic processes or non-generalizable patterns perturbing the data beyond the behavior the tree learned. As a result of this, algorithms must be careful to train only until the tree meets the expected specifications in regards to accuracy and other performance metrics. In addition, pruning techniques to remove non-generalizable branches are employed. When this issue is taken into account, CARTs act as a good baseline algorithm. This has resulted in them being used in many ensemble-based methods, which use multiple copies of an algorithm to produce results. Ensemble-based methods will be discussed in further detail in Section 2.3.1.

While CART is a simple and useful method, the recent resurgence of Machine Learning as a useful field has been driven by extensions of the multilayer perceptron (MLP) [31], also known as the feedforward neural network (FFNN). An MLP is composed of layers. Each layer is itself composed of nodes, each of which holds a value. The inherent meaning of a node value depends on what type of layer the node belongs to. There are three different types of layers: input layers, hidden layers, and output layers. For each MLP there is one input layer, which is composed of inputs x_i. Each hidden layer node is a linear combination of all the nodes from the layer before it. How much each node from the previous layer impacts the current node

is defined by a set of weights and biases. After this linear combination, the node applies a nonlinear transform function, often called the activation function. There are a variety of possible activation functions, each of which is more applicable to certain data conditions. Frequently, this transform is used as a squashing function to limit the outputs to a certain range [32]. Some examples of this are the tanh() function, which limits the range to (-1,1), and the logistic function, which limits the range to (0,1). Non-linearities are introduced to allow the MLP to represent more complex relationships. Without the activation function, adding additional layers would not result in any increase in representational ability. The output layer takes the values from the last hidden layer and, after one more linear combination, outputs the MLP's prediction of the target value.

In order to train an effective model, there needs to be a way to evaluate how effective the model is at any given moment. For an MLP, this is accomplished by using a cost function J, which is used to evaluate how well the model is performing compared to the ground truth. In the context of a classification problem, a commonly used cost function is cross-entropy, also known as log loss. Cross-entropy, a concept taken from information theory, represents the average number of bits needed to identify if a sample was taken from probability distribution p or probability distribution q. By choosing $p = y_{truth}$ and $q = y_{pred}$, cross-entropy can be used as a distance metric. A commonly used cross-entropy based cost function is described in Equation (2.1) [33], where y is a binary indicator, with 0 and 1 representing the two different classes that the model is choosing between. By using this formulation, the only factor to loss is the class predicted.

$$J_{crossEntropy} = \frac{\sum_N (y_{truth} log(y_{pred}) + (1 - y_{truth}) log(1 - y_{pred}))}{N} \tag{2.1}$$

This error function is based on a binary classification problem, but can be easily

extended to a multi-class problem by using a separate binary classification for each class, and sum the cross-entropic error for each class to get the total cross-entropic error.

In the context of a regression problem, a commonly used cost function is mean squared error (MSE). The definition of MSE is given in Equation (2.2) [33]. As the name implies, it represents the mean of the squared error between the y_{pred} and y_{truth}. Unlike Equation (2.1), y here represents a continuous value, so the distance of the prediction to the truth value impacts the metric.

$$J_{MSE} = \frac{\sum_N (y_{pred} - y_{truth})^2}{N} \tag{2.2}$$

By using these cost functions, a clearer picture about the effectiveness of a model is created. With training reframed as an optimization problem, the simplest solution is to use gradient descent, in which the partial derivatives of the cost function are used to adjust the weights towards a local minimum value of the cost function.. Upon taking the gradient of the cost function, the resulting value will be negative for downward slopes and positive for upward slopes. By updating the weights in the opposite direction of the gradient, the resulting loss value from the updated network will move towards a minimum. This is represented in Equation (2.3) [34], where h represents the value to change the weights W, and J is the Jacobian (the matrix of all first-order partial derivatives of a vector) of the objective function. This method converges well with simpler objective functions, but has the potential to converge slowly, depending on the α parameter and the Jacobian.

$$h_{gd} = \alpha J^T W (y_{truth} - y_{pred}) \tag{2.3}$$

Another solution is called the Gauss-Newton method [35], which is applicable to a sum-of-squares objective function, like J_{MSE}. It assumes that the objective

function is approximately quadratic near the solution, then uses a first-order Taylor series expansion to approximate the Hessian of the objective function to be J^TWJ. This then allows for the weight update procedure to follow Equation (2.4). This converges much faster than gradient descent for moderately-sized problems, but is only applicable to a stricter set of objective functions, as it depends on the function being twice differentiable. It also doesn't provide as much of a speedup when the first derivative of the objective function has repeated roots.

$$h_{gn} = (J^TW(y_{truth} - y_{pred}))(J^TWJ)^{-1} \tag{2.4}$$

The solution that is frequently used (and is the MATLAB default in training a neural network) is the Levenberg-Marquardt (LM) method [36]. This is effectively a combination of the two prior optimization methods. It is described in Equation 2.5.

$$h_{lm} = (J^TW(y_{truth} - y_{pred}))(J^TWJ + \lambda \cdot diag(J^TWJ))^{-1} \tag{2.5}$$

The variable λ represents the parameter that controls how similar the update is to Gradient Descent, and how similar the update is to Gauss-Newton. A large λ makes LM more similar to Gradient Descent, and a small λ makes LM more similar to Gauss-Newton. The variable λ is usually initialized to be large, to enable the first updates to be small, as a result of Gradient Descent's slower convergence. If an update increases the objective function, λ is increased, moving closer to Gradient Descent. Otherwise, λ decreases, moving closer to Gauss-Newton. This is reasonable because as the weights approach the optimal solution, it is more reasonable to assume that the cost function is quadratic.

For an algorithm like linear regression, one of these update methods by itself is enough as an update procedure. However, MLPs require a modification of this

process. This is due to the multiple layers involved, which introduce additional complexity with the interconnections between layers. The process of updating MLPs is called Backpropagation. Gradient Descent Backpropagation will be described in Algorithm 2, as it is the simplest to understand. Backpropagation as applied to Levenberg-Marquardt can be found in [37].

Algorithm 2 Pseudocode for Backpropagation

1: Given: training set X.
2: **procedure** BACKPROPAGATION
3: Set input activation $a^1 = \sigma(X)$
4: **for** $l = 2{:}L$ **do**
5: $z^l = w^l a^{l-1} + b^l$
6: $a^l = \sigma(z^l)$
7: $\delta^L = \nabla_{a^L} C \odot \sigma'(z^L)$
8: **for** $l = L - 1{:}2$ **do**
9: $\delta^l = (w^{l+1}\delta^{l+1}) \odot \sigma'(z^l)$
10: Output 1: $\frac{\delta C}{\delta w^l_{jk}} = a^{l-1}_k \delta^l_j$
11: Output 2: $\frac{\delta C}{\delta b^l_j} = \delta^l_j$

The symbol $\odot$ is the Hadamard Product, which is an element-wise product of two vectors with the same length. L is the number of layers in the MLP. z^l is the weight and bias values at layer l, and a^l is z^l passed through the activation function $\sigma(z)$. Both of these are vectors, with each element representing one node in the layer. C is the cost function, used to evaluate the difference between the predicted value and the truth value. δ^l is the error at layer l.

The backpropegation algorithm used in MLPs can be broken into 4 steps: forward propegation, backpropegation at the last layer, backpropegation at the hiden layers, and updating of the weights. Forward propegation simply applys the training inputs to the network as normal (represented by steps 2-6). This involves applying the weights and biases to the output of the previous layer, and then applying the activation function. Then, the backpropegation error at the output is calculated, using the gradient of C relative to a^L. (step 7). The error is carried back through

the network in the same way that the MLP is applied except backwards, passing the error of the previously calculated layer though the weights of the current layer. The bias values can be ignored due to the derivative that is taken. Once the error propagation is complete, the partial derivatives of each node can be calculated, and can be used in a gradient descent manner to modify the nodes.

In recent times, improved versions of backpropegation have been proposed and utilized, such as RMSprop [38] and Adam [38]. Both of these improvements are extensions of standard gradient descent backpropegation. Standard backpropegation utilizes one learning rate for all weight updates. This rate doesn't change during training. RMSprop, also known as Root Mean Square Propagation, maintains per-parameter learning rates that get adjusted by a moving average of the recent gradient values for the individual weight [38]. Adam takes this a step further and adapts the parameter learning rates based on the gradient's variance, in addition to the gradient's mean that RMSProp uses. Doing so results in quicker convergence than standard backpropegation. Details can be found in [39]. In modern times, Adam has been frequently recommended as the default optimization method in training neural networks [40]. Both RMSProp and Adam are still based on first-order methods, unlike Gauss-Newton and Levenberg Marquardt, which is likely one of the reasons why the new algorithms have surpassed the old ones in usage. Nonetheless, the same concern in online usage remains. This thesis will stick to LM, as the project it is building on top of is based off of LM.

2.1.2 Reinforcement Learning

While both supervised and unsupervised learning are built on the premise of identifying some form of structure from a given set of data, the goal of Reinforcement Learning (RL) is fundamentally different. The core premise of RL is of an agent trying to learn an environment. In passive RL, there agent has no influence on the

environment, and can just observe some reward based on the location it is at. In active RL the agent is able to take actions, and so the reward is based on the location and the action that was taken. With this reward value, the agent can update its understanding of the environment, in a form called the state-transition model. How it does this depends on the RL algorithm being used. Active RL is more relevant to this thesis and will be the primary focus of this section.

While there are multiple different ways to pose an RL problem, the one that is most applicable to the problem at hand is the Multi-Armed Bandit. In this model, an action set has multiple reward functions, used as metrics for how well a task was executed. The agent is the "bandit", trying to get the "best" reward overall over the multiple reward functions. How best is defined is difficult, as the interaction between reward functions is not straightforward, and the impact of each reward function may vary. Reward functions serve a similar purpose to the objective function in supervised learning, but are not required to be differentiable.

Another potential model is that of a state-transition problem, modeled as a Markov decision process. Rephrased in an RL context, the target problem is changing radio parameters so that optimal performance is achieved in context of the current environmental conditions. There are a wide range of conditions that affect the state-transition model, including the communications channel. This channel is affected by the dynamic geometry of the line-of-sight between the transmitter and receiver, as well as atmospheric and space weather. These complex and changing conditions make finding the state-transition model and action-state mapping analytically an intractable pursuit.

This difficulty is what makes RL a good fit for tackling the problem. RL makes few assumptions about the behavior of the process being studied, other than the fact that it can be reasonably modelled by a state-transition model. In addition, RL is designed to be constantly used and updated, instead of waiting for the optimal

solution to be calculated before running. With these two properties, the primary concern shifts from intractability to ensuring the agent interacts with the environment in an efficient way to find the best possible policy. The best way to form a policy would be to evaluate the action-value function $Q(s, a)$ at each possible state and action. However, this becomes intractable quickly. Instead, the agent periodically explores policies, with either on-policy or off-policy approaches. On-policy approaches affect the policy that the agent uses to make decisions, while off-policy approaches affect a separate policy from the one that the agent uses to make decisions. Since the target platform is a cognitive engine that actively adapts to the environment conditions, the focus will be on-policy approaches.

The model-free method that is relevant to this thesis is Temporal-Difference (TD). This method updates $Q(s, a)$ using the past experiences at each time step. This makes it useful for time-sensitive applications. When used on-policy, it is called State-Action-Reward-State-Action (SARSA). The pseudoalgorithm for SARSA is provided in Algorithm 3.

Algorithm 3 SARSA Pseudoalgorithm

procedure SARSA
    Initialize $Q(s, a)$ for all states s and actions $a \epsilon \mathcal{A}(s)$ arbitrarily. Set $Q(terminal\ state) = 0$.
    while Learning **do**
        Initialize s
        Choose a from s using policy derived by Q.
        while not terminal state **do**
            Take action a, observe reward r and next state s'.
            Choose a' from s' using policy derived from Q.
            Update Q: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$.
            Update $s \leftarrow s'$,$a \leftarrow a'$.
    return Updated Q table.

In the SARSA algorithm described in algorithm 3, α is the learning rate of the algorithm, r is the reward, γ is the discount factor for future rewards. In our context, there is no terminal state. The algorithm instead ends when there is no

longer a connection between the SCaN testbed and the ground station. Described briefly in words, the SARSA algorithm is as follows. Before running, the Q function is arbitrarily initialized for all possible states s and actions a. Then, the state is initialized to its initial state. From this, a is chosen based on the Q-value table. A reward r and next state s' are then observed. Next action a' is then chosen from the Q-value table using s'. The value of the Q-value table for the current state and action $Q(s, a)$ is then updated. The value within the brackets is known as the TD error, and is the difference between the current value of $Q(s, a)$ and the updated prediction which adds the observed reward from taking the action to the Q-value from the next state-action pair s' and a'. From this point, s advances to the next state s', and the action to be taken a is replaced by a'. From here, the process of taking actions and updating the Q-value table ris repeated.

In the context of cognitive radio, only the immediate reward (i.e.$\gamma = 0$) is relevant [2]. In addition, any action can be taken from any state without the need for planning. This results in a modified Q function:

$$Q_{k+1}(s_k, a_k) = Q_k(s_k, a_k) + \alpha[r - Q(s_k, a_k)]. \tag{2.6}$$

Based on $Q(s_k, a_k)$, a knowledge base called a Q-table is built up. In this table, previous Q-values are mapped to the state-action pairs that resulted in the Q-values. In SARSA, the value of $Q(s_k, a_k)$ is updated when the action is taken. Beyond $Q(s, a)$, a reward function $r = g(s, a)$ and an exploration policy $a = h(s)$ need to be defined before a practical model is complete. In the context of the CE, a multitude of functions are relevant to the overall performance of the system. These performance functions may or may not have conflicting responses. The reward function mapping environmental information to performance is a combination of values from these performance functions. The overall result is then represented as a percentage of the

maximum possible performance value.

One of the most important tradeoffs for RL is deciding how much time should be spent exploring the environment versus how much time should be spent exploiting the knowledge already gathered. This is frequently called the Explore-Exploit tradeoff [41]. In the context of SARSA, exploring would represent choosing an unvisited action while exploiting would be choosing an action that has the highest Q value. The exploration policy is responsible for making this decision. This tradeoff is important because exploring too much would create a detailed understanding of the environment but would not actually use the information, while exploiting too much could be using subpar information as the environment is only minimally explored. The preventing of either of these cases becomes an important issue.

There are many different approaches to tackling the explore-exploit tradeoff. The simplest one is to use an ϵ-greedy policy [41]. In this method, an random numbe rn is drawn form $\mathcal{U}(0,1)$. This is compared to $\epsilon \in (0,1)$. If $n < \epsilon$, a random action is chosen. Otherwise, a greedy action based on the Q-table is chosen. ϵ can be set as a function decreasing over time. This results in more random actions being chosen at the beginning of the running the algorithm and more exploitation of the explored environment as time goes on. Other exploration policies include value-difference based exploration (VDBE) [42], which changes ϵ based on TD error, as well as Boltzmann exploration and probability matching [43]. In this work, a time-varying ϵ-greedy policy is used.

2.2 Previous work on NASA SCaN Testbed

This section first describes the problem that is trying to be solved by the NASA John H. Glenn Research Center (GRC) Space Communications and Navigation (SCaN) Testbed project [44]. With this foundation, the architecture of the cognitive engine (CE) that was developed will be described as well. Finally, the primary goals of the

extension that this thesis is focused on will be described.

2.2.1 Problem Statement

The Cognitive Engine project, and by extension this thesis, is built on top of the SCaN Testbed, an experimental communication system developed by NASA to research implementation solutions for issues related to SDR-based communications to and from space [1]. The platform consists of multiple SDRs that are configured to operate at S-band and Ka-band, both in direct communications to ground stations on Earth and in communication with NASA's satellite relay infrastructure named Tracking and Data Relay Satellite System (TDRSS [45]). For this thesis, communication with ground stations is the central focus. This platform is used to research real-world satellite dynamics between spacecraft and relay satellites or ground stations. Some of the dynamics studied include time-varying Doppler changes, differences in thermal conditions, interference, range variation, ionospheric effects, and other impairments to propagation.

The SDRs chosen for the SCaN testbed are flight-grade systems, fully compliant with NASA's Space Telecommunications Radio System (STRS) [46] SDR architecture. This architecture provides abstraction interfaces between the radio software and proprietary hardware, allowing for third-party software waveforms and applications to interact with and run on the radio. STRS also provides a library of waveforms available that provide various modulation, coding, framing, and data rate options. This work utilized the waveform based on the DVB-S2 standard.

On top of this platform, solutions such as adaptive communications using cognitive decision making can be researched, which will help solve communication issues on Earth as well as allow for the development of space communication systems that will enable space exploration in the near future [47]. SDRs are important in this development, as their flexibility and reconfigurability provides a strong tool that

enables a wide variety of tests. However, SDRs tend to be more complex than traditional fixed-configuration radios. Cognitive Radio (CR) systems help reduce the complexity and risk in using these systems. The work that this thesis is extending plays a part in this effort, providing CR algorithm research for SDR systems in space.

Three areas have emerged as candidate application areas for CR: node-to-node communications, system-wide intelligence, and intelligent internetworking [48]. The first application focuses on radio-to-radio links between mission spacecraft and a ground terminal [48]. Cognitive decisions may improve throughput across the communication link by adjusting waveform settings to maximize user data and symbol rate, while using more conservative settings during parts of the pass with poor channel conditions. The second application is system-wide intelligence, where the CR systems make operational decisions normally performed by operators [48]. For instance, CR systems could be applied to scheduling, asset utilization, optimum link configuration and access times, fault monitoring, and failure prediction, among others. In this application, a CR system could reduce operation oversight and reduce operational complexity, by choosing among the large search space of configurations. The final application of CR systems applies to the network level [48]. With control and data functions of the communications network, CR systems could optimize data throughput using QoS metrics such as bit error rate. Learning network behaviors could enable the CR to provide throughput and reliability benefits.

For this thesis and the work preceding it, the first application was considered to be the main priority without loss of generality. Regardless of the application, the need for verification and ground-testing of all operational conditions before launch is apparent, since doing so minimizes the risk of malfunction on-orbit. Once this verification is complete, tests (including the one in this thesis) can be conducted in order to more properly study the impact of CR systems on communications.

The project this thesis is extending, titled "Intelligent MAC protocol for SDR-based satellite communications", involved the NASA Glenn Research Center (GRC) such that cognitive engine algorithms for future space-based communications systems can be developed. This effort included the opportunity to perform on-orbit tests with the SCaN Testbed SDRs. This effort's R&D team is composed of members from WPI and the Pennsylvania State University, in collaboration with the SCaN team at NASA GRC. The project commenced in November 2014, and the initial tests were completed in May 2017. This extension is scheduled to be completed by December 2018.

During the course of this project, the team designed and developed a proof-of-concept of an intelligent Medium Access Control (MAC) protocol to maximize data link performance and improve robustness of space communications systems. This uses SDRs for low margin data links operating on dynamically changing channels. The protocol's core is composed of a Cognitive Engine (CE) that balances multiple different objective issues during radio-resource allocation. Radio parameters may be adapted to mitigate channel impairments or when meeting new performance requests. Key capabilities for cognition are prediction and learning techniques assisted by third-party databases when they are available.

This CE project aligned specifically with NASA's Communication and Navigation Systems Roadmap Technology Area 5 [49], which focuses on the concept of a cognitive radio in space that senses the environment, autonomously determines when there is a problem, attempts to fix it, and learns about the environment as they operate. It also acts as a step towards reducing the limits that communication techniques impose on future missions and their critical phases. The CE algorithm, as well as expreiment results, will be used to assess the performance of the adaptive MAC protocol performance for on-orbit SDRs, and will help in the design of future MAC protocols and mitigation techniques that utilize intelligent radio parameter

reconfiguration.

There are four different communication channels between the fixed ground stations at GRC/White Sands, the moving ScaN Testbed SDR on board the ISS, and the TDRSS satellite at a GEO orbit, as illustrated by Figure 2.3. Radio parameter reconfiguration might be done during periods of signal fading, especially those happening during low elevation angles of the SCaN Testbed antenna, while tracking the TDRSS satellite, or low elevation angles from a GRC ground station antenna, while tracking the ScaN Testbed. The project focuses on the direct communication between GRC and the SCaN Testbed.

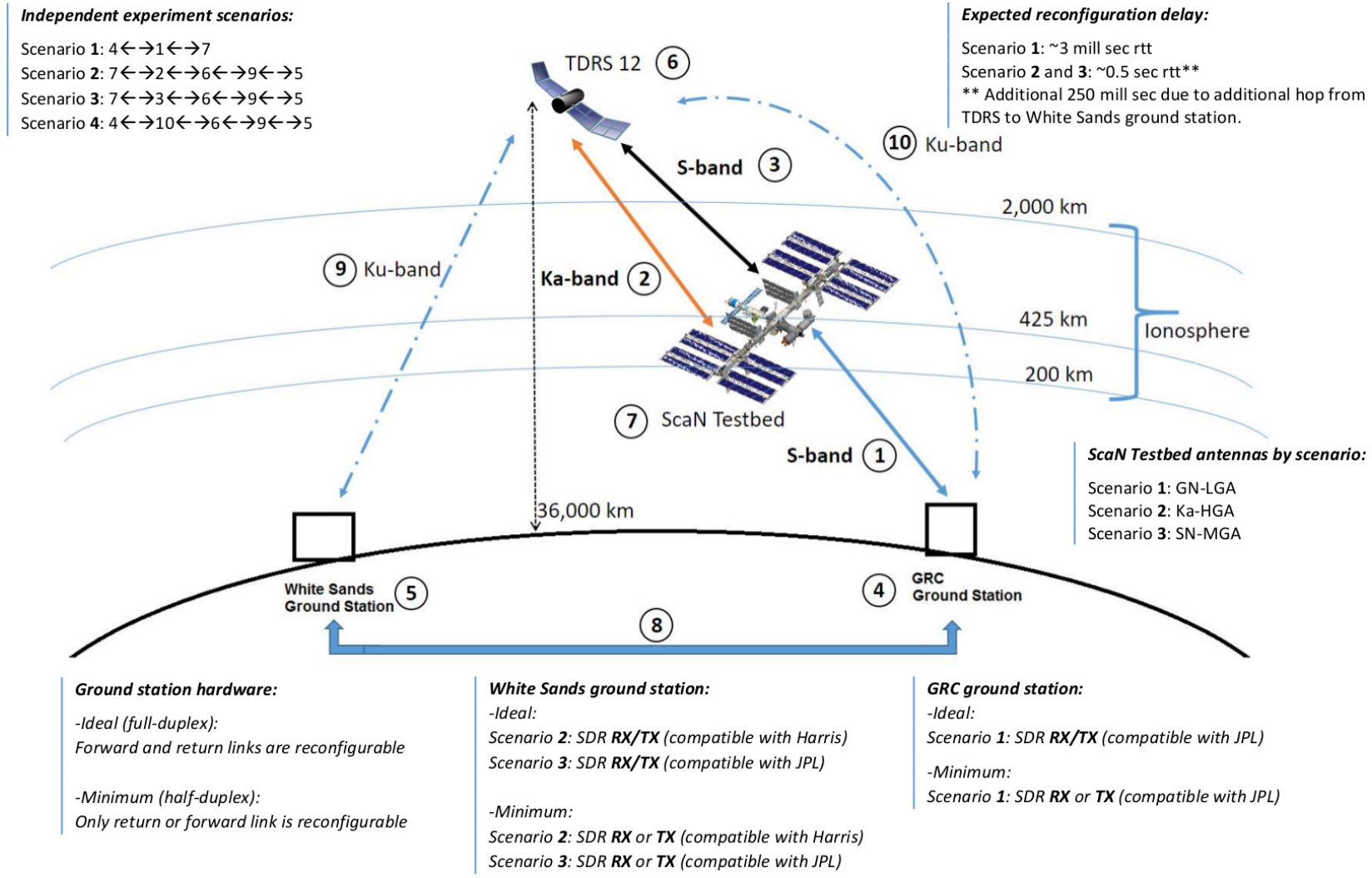


Figure 2.3: Four potential communication channels which the CE could be applied. Scenario 1 with half-duplex was implemented in [1], and was used in this thesis.

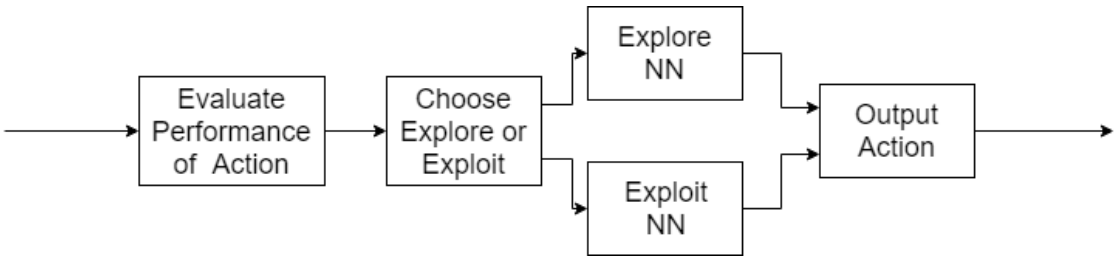


Figure 2.4: A high level overview of the proposed CE. After an action was transmitted and the response was received, performance metrics are observed and combined into a fitness score (as described in Chapter 3). Using this fitness score, the CE exploits its knowledge of the environment or performs a guided exploration. The resulting action is then transmitted to the SDR.

A high level overview of the proposed CE is shown in Figure 2.4. The platform gathers telemetry data reported from the transmitter or measured at the receiver. A predictor employs past information to predict which radio parameter values will achieve good performance. Learning techniques are used for prediction in order to control storage memory growth. Decision logic based on the predictions will choose what adaptations will need to occur. The CE learns the environment behavior by building a model that maps observed telemetry data to radio parameter sets.

In this thesis, there were two primary goals: To explicitly deal with the catastrophic forgetting that occurs in MLPs, and to investigate the potential utility of Generative Adversarial Networks (GANs). In the following section, both of these subjects will be discussed in more detail.

2.3 Advanced Topics in Machine Learning

In this section, a couple relevant complex machine learning topics will be discussed. The first subsection discusses the online learning paradigm and the issue of Catastrophic Forgetting that is associated with using connectionist networks for this purpose. This is followed by an overview of Generative Adversarial Networks (GANs), and an investigation into the applicability of the concept to the CE.

2.3.1 Online Learning and Catastrophic Forgetting

In supervised learning, there are two different learning paradigms: online learning and offline learning. Offline algorithms assume that there is one set of inputs and outputs to train on, and that once training is complete there will be no updates to the algorithm. The training methods described in Section 2.1 are all considered to be offline methods. This is contrasted with online algorithms, which update as new data is received. The choice between an offline algorithm and an online algorithm is dependent on the problem that is selected. With a stationary dataset, a single training period can be sufficient. However, if the dataset is nonstationary, there is a reason for multiple training periods capturing the shifting of the data, also known as concept drift [50]. Since effective space communications systems are dependent on many nonstationary variables, online methods are more suited for this project.

The most basic category of online methods consists of generalizations of offline methods. In this category, a window of incoming data is collected. Once this window is full, the collected data is used to train using an offline method. Then, some number of datapoints are dropped and the window is filled again. This moving window process is the most straightforward extrapolation of offline algorithms. In the baseline CE, this method is applied to LM [1]. The moving-window approach has one major problem that it introduces. The LM algorithm updates the weights using

the data that is put into itwhich is similar to many first-order offline algorithms (Stochastic Gradient Descent [51], Adam [39], RMSProp [38], etc.). In this type of online method, the window moves and overwrites old datapoints, with these old datapoints no longer influencing how the model is trained. As a result of this, the model will lose the information that was encoded as a result of these datapoints; this is called Catastrophic Forgetting [52]. Catastrophic Forgetting primarily affects learning in non-stationary environments. If an environment is stationary, the fact that datapoints are no longer used in training has little effect as the datapoints replacing them will come from the same probability distribution. However, in non-stationary environments, the loss of information is more significant. If there is any cyclical nature of the changes, the information will have to be relearned each time.

One way to deal with this is to use a recursive training method, which updates the network based on batches of samples each time but uses these samples to update an internal state. The recursive implementation LM, as described by Ngia and Sjoberg [53], is used in this thesis. In this process, there is a forgetting factor α built in that allows for the network to forget past inputs in a more explicit manner than the standard batch-update method. In the following equations, $\varepsilon(h_t) = \sqrt{J(h_t)}$, P_t can be considered the covariance matrix of the weights, $\Omega^T(h_t)$ is a modified gradient for y, allowing for a normalization factor ρ to be added in one weight at a time. The second row of $\Omega^T(h_t)$ is 0 for all indices where $i \; mod \; (nWeights + 1) \neq t$, and 1 where $i \; mod \; (nWeights + 1) = t$. α is the rate of forgetting.

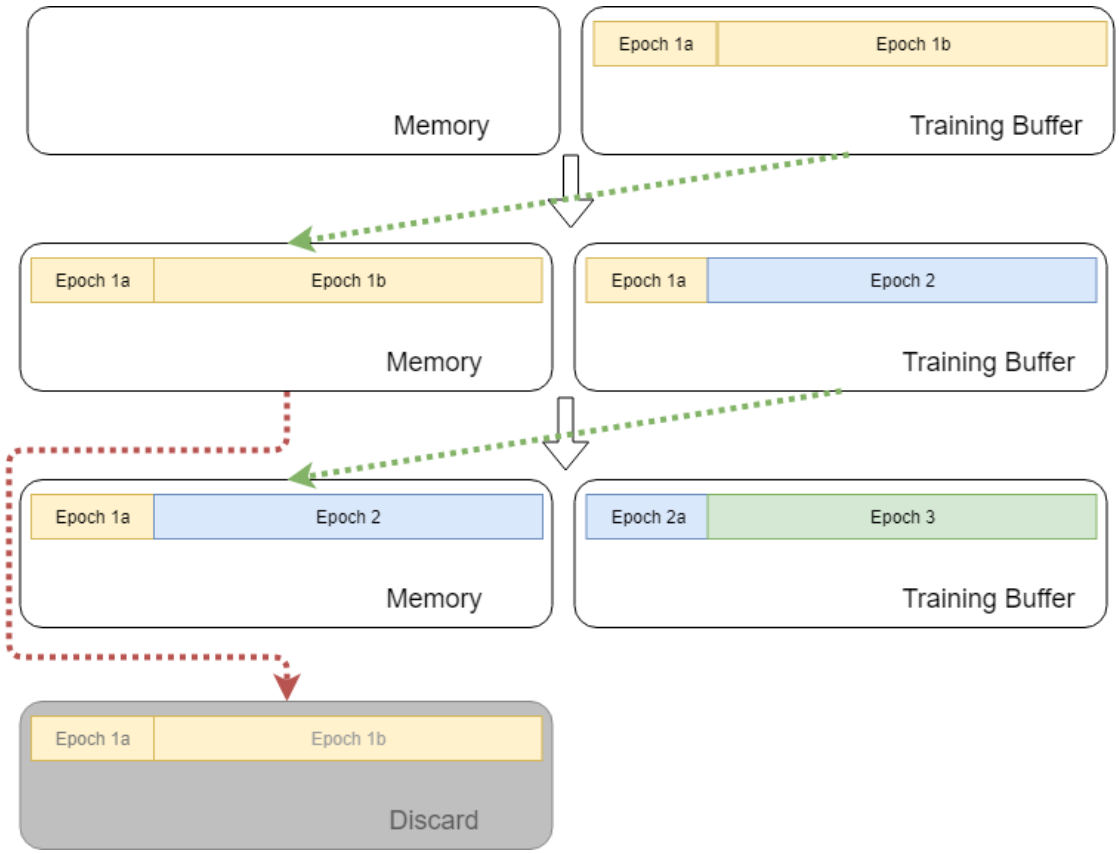


Figure 2.5: A visual representation of how catastrophic forgetting occurs. The first epoch is before any training occurs. All the data in the training buffer is encoded into the MLP. As the second epoch is collected, a portion of the data in the training buffer is kept from the first epoch. All other data is overwritten by new samples. Retraining with this new buffer overwrites the encoded information from the first buffer.

$$\Omega^T(h_t) = \begin{bmatrix} & \nabla_y^T(h_t) & \\ 0 \quad \ldots & 1 & \ldots \quad 0 \end{bmatrix} \tag{2.7}$$

$$\Lambda_T^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & \rho \end{bmatrix} \tag{2.8}$$

$$S(h_t) = \alpha_t \Lambda_t + \Omega^T(w_t) P_{t-1} \Omega(w_t) \tag{2.9}$$

$$P_t = \frac{1}{\alpha_t}[P_{t-1} - P_{t-1}\Omega(h_t)S^{-1}(h_t)\Omega^T(h_t)P_{t-1}] \tag{2.10}$$

$$h_{t+1} = h_t + P_t \nabla_y(h_t)\varepsilon(h_t) \tag{2.11}$$

The update process is split into three substeps, each substep updating a a related aspect of the internal or external state of the MLP. The first step computes an intermediate state $S(h_t)$, which reduces the necessary matrix inversion to a 2×2 matrix. The second step updates the covariance matrix, and finally the third step updates the weights.

A different approach for dealing with catastrophic forgetting is to use ensemble training methods [54]. Ensemble methods use a collection of learners to produce improved results over any individual learner by itself. In general, ensemble methods use a set of weaker algorithms that are combined with or without some sort of weighting to produce a unified result In classification problems, a weighted majority voting is often used while in regression problems a weighted median or mean can be employed. Frequently, a CART is used as the base algorithm due to its simple implementation and reasonable performance. However, any algorithm can be selected. Once a base algorithm is chosen, the difference between ensemble implementations is how the ensemble is created and used. For instance, in Adaboost [55] a series of base learners is trained, where each successive base learner is trained in a way that more heavily weights points that the previously trained learners have trouble with. This differs from bagging [56], a different algorithm where each base learner is trained with a different subset of the total training data. These examples are provided simply to show how ensemble learning methods can differ.

The method most relevant to addressing the problem of catastrophic forgetting is Learn++.NSE [57]. For each window of samples, Learn++.NSE creates a new learner that is trained on the window. The samples in the training window are weighted by how much trouble the pre-existing learners had in predicting the value, ensuring that the new learner is better at recognizing these samples. It then weights the new learner and existing learners based on how well they work on the training data, using a metric such as MSE. In doing this, existing learners contribute to the

resulting prediction based on how well they work on the most recent window of data.

Algorithm 4 Learn.NSE++ Pseudoalgorithm

procedure LEARN.NSE++
  $t = 1$, $numActiveClassifiers = 0$
  m^t is the data window size at epoch t.
  while Running **do**
    if $t = 1$ **then**
      for $i = 1 \rightarrow numActiveClassifiers$ **do**
        $D(i) = w(i) = 1/m^1$
    else
      Apply data window to current ensemble.
      $E^t = \sum^{m^t}(1/m^t) * squaredError$(current learner, current data)
      At this point, E^t should be a vector with size m^t.
      for $i = 1, i < numActiveClassifiers, i++$ **do**
        Update learner weights:
        $w_i = 1/m^t * \sum squaredError$(current learner, current data)
      Normalize learner weights: $D = w \odot (1/\sum w)$
    if $numActiveClassifiers = numClassifiers$ **then**
      Remove oldest classifier.
    else$numActiveClassifiers = numActiveClassifiers + 1$
    Train new Learner on new data window.
    Evaluate all existing classifiers with new data.
    for $i = 1 \rightarrow numClassifiers$ **do**
      $\epsilon_i^t = \sum D(i) * squaredError($ current learner, current data)
      $\beta_i^t = \epsilon_i^t/(1 - \epsilon_k^t)$
      Compute average of normalized error for h_i:
        $\omega_i^t = 1/(1 + exp(-a(t - i - b)))$
      $\omega_i^t = \omega_i^t / \sum_{j=0}^{t-i} \omega_i^{t-j}$
      $\hat{\beta}_i^t = \sum_{j=0}^{t-i} \omega_k^{t-j} \beta_i^{t-j}$
      if $t > numClassifiers$ **then**
        Calculate classifier voting weights $W_i^t = log(1/\hat{\beta}_i^t)$
    Training Done.
    while Filling training buffer **do**
      get Hypothesis $H(x) = \sum_{i=0}^{numClassifiers} W_i^t * h_i(x)$
    Buffer full, $t+ = 1$.

In the standard implementation of Learn++.NSE, a classification problem is targeted, and it is supposed that an infinite number of learners can be created.

The adjustment from classification to regression requires the algorithm to use MSE instead of cross-entropy as an error function, and a method for limiting the number of learners is required as well. Due to time constraints, a simple pruning process of choosing a maximum number of learners and replacing the oldest learner was chosen. More complex pruning methods have been explored in [58].

2.3.2 An Investigation of GANs and Their Utility to Space Communications

One of the major goals of this extension to the previous work is to evaluate whether the concept of Generative Adversarial Networks (GANs) [59] was applicable to the project. Like the CE in its current state, GANs are composed of two neural networks providing complementary data. In this section, GANS will be described in more detail, as well as their applicability to the CE.

The primary purpose of a GAN is to implicitly model some probabilistic data distribution. This is usually done in the case where data is either too complex to model effectively or completely unknown. Prior work has focused primarily on image-based data. Common datasets to be modeled for testing include MNIST (hand written digits), CIFAR-10 (natural images) and the Toronto Face Dataset (TFD) [59]. These distributions happen to be all image-based, but there is no explicit need for the data distribution to be an image. Images just happen to be an extremely complicated representation space with many nonlinear patterns, making it a good data type for practice.

A GAN is composed of two separate learners: a Generator($\mathcal{G}$) and a Discriminator($\mathcal{D}$). $\mathcal{G}$ and $\mathcal{D}$ are configured to be in competition,where $\mathcal{D}$ is attempting to distinguish between points that are generated by $\mathcal{G}$ and points that are sampled from the true distribution. $\mathcal{D}$ returns a 0 if it thinks the datapoint is not from the true distribution, and a 1 if it thinks the datapoint is. $\mathcal{G}$ is attempting to fool $\mathcal{D}$ into confusing its

generated datapoints with the data from the true distribution. Its best case scenario is when $\mathcal{D}$ is only correct around 50% of the time, as this makes $\mathcal{D}$'s prediction as good as randomly guessing which distribution the sample is from.

In a more formal sense, $\mathcal{D}$ can be considered as dealing with a standard classification problem, attempting to classify whether or not a sample it is given is from the real distribution or the artificial one. $\mathcal{G}$ can be considered as dealing with a regression problem, trying to map a random uniform input $U(0, 1)$ to the true distribution.

Figure 2.6 shows an overview of the GAN architecture. The task of $\mathcal{G}$, which is modeling a probability distribution, is more difficult than the task of the discriminator, $\mathcal{D}$, classification of incoming data. As such, when the discriminator ceases to improve, it is often frozen and only $\mathcal{G}$ gets updated.

The training of GANs is based on a value function $V(\mathcal{G}, \mathcal{D})$, which is dependent on both the generator and the discriminator. Training the GAN accomplishes the following:

$$\max_{\mathcal{D}} \min_{\mathcal{G}} V(\mathcal{G}, \mathcal{D}),$$

where:

$$V(\mathcal{G}, \mathcal{D}) = E\{p_{real}\} log(\mathcal{D}(x)) + E\{p_{generated}\} log(1 - \mathcal{D}(x)). \tag{2.12}$$

The competing nature of $\mathcal{G}$ and $\mathcal{D}$ is shown by the fact that one is trying to maximize the value function while the other is trying to minimize it. The first part of $V(\mathcal{G}, \mathcal{D})$ represents the log-probability that the discriminator successfully classifies real datapoints as real. The second part represents the log-probability that the discriminator successfully classifies generated datapoints. $\mathcal{D}$ is thus trained by ascending the gradient of $V(\mathcal{G}, \mathcal{D})$, and $\mathcal{G}$ is trained by descending a modified version of the gradient. Since $\mathcal{G}$ is only able to affect the probability of $\mathcal{D}$ providing

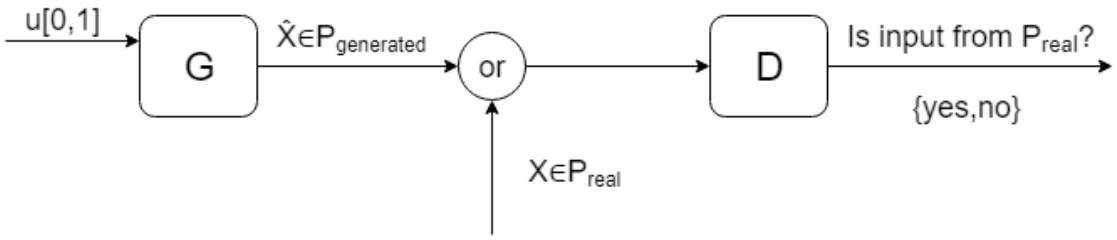


Figure 2.6: A flow diagram describing how GANs are structured. $\mathcal{G}$ attempts to map a uniform random variable to the dataset of real samples. $\mathcal{D}$ attempts to distinguish the samplse generated by $\mathcal{G}$ from the real samples.

a false positive, it only attempts to minimize this part of $V(\mathcal{G}, \mathcal{D})$. Consequently, this becomes:

$$\min_{\mathcal{G}} E\{p_{generated}\} log(1 - \mathcal{D}(x)). \tag{2.13}$$

In order to use a non-saturating criterion, the minimization is rearranged to be a maximization:

$$\max_{\mathcal{G}} E\{p_{generated}\} log(\mathcal{D}(x)). \tag{2.14}$$

Beyond the value function, standard gradient-based backpropagation methods are used for updating the networks. Both networks are trained in tandem: a batch of real samples is collected, a batch of generated samples are created, and then the results of feeding these samples through $\mathcal{D}$ are used to update both $\mathcal{D}$ and $\mathcal{G}$.

While GANs are a versatile method for modeling a data distribution, the training process is far from robust. The first issue is that there is no guarantee that the pair of models will converge. Due to the adversarial nature of the network, the convergence of a GAN depends on both models converging with competing objectives. The standard GAN procedure does not have any guarantees that this will happen. Another issue is mode collapse, when $\mathcal{G}$ starts to produce very similar samples for different inputs. This may result in good values for $V(\mathcal{G}, \mathcal{D})$, but will only model a

specific subset of the real data distribution. A final issue is that the loss value of $\mathcal{D}$ can quickly converge to zero, making there no reliable way to update $\mathcal{G}$. This is called the vanishing gradient problem [60], and is not specific to GANs.

In practice, there are a couple of approaches that are used to reduce the impact of these issues. One method is to normalize the inputs of each layer by subtracting the input mean and dividing by the input standard deviation [61]. This ensures that inputs of vastly different magnitudes can be represented in a more similar manner. Doing this introduces a "standard deviation" parameter and a "mean" parameter for each node, allowing for the training process to use "denormalized" nodes if it is optimal without reducing the overall stability. This most directly affects the vanishing gradient problem, but also has positive effects on convergence rate as well.

An approach used to deal with mode collapse is mini-batch discrimination [62], which adds an input to the discriminator that encodes the distance between a sample in a mini-batch and the other samples. This makes the discriminator able to tell if the generator is producing the same outputs. Another approach, called feature matching [63] which alters the goal of $\mathcal{G}$ in order to try to match an intermediate activation of $\mathcal{D}$ from real samples with its generated samples. The additional information makes it more able to represent complex representations. Heuristic averaging [63], which penalizes network parameters if they stray from a running average of previous values, is a useful approach to aid in making the GAN converge.

Beyond the heuristic approaches, there have been a few attempts at alternative, more robust formulations of GANs. The most prominent approach is the WGAN [64]. This modifies the cost function of the GAN to use Wasserstein distance, instead of cross-entropy. The intuition of the Wasserstein distance measure is that:

- P_x and P_y are similar to piles of earth.
- $\gamma(x, y)$ is the amount of "mass" that needs to be moved to make P_x into P_y.

- Wasserstein distance is the "cost" of the optimal transport plan of γ.

By using this formulation, the authors in [64] found that the resulting GANs were more robust than the standard version.

In addition to making GANs more robust, work has gone into combining the concept with other ML and DL architectures. Some of the interesting alternate GANs are DCGAN [65], which introduces Convolutional Neural Networks (CNNs) to the GAN, and Adversarially Learned Inference (ALI), which introduces a network that encodes the data into a latent data space that can be used in a variety of ways [66].

On a surface level, the CE has some similarities to the GAN. As with a GAN, the CE uses two neural networks working in combination to produce different types of results. However, many of the similarities end here. The performance of the explore and exploit networks are not particularly related, as the performance of one does not indicate the performance of the other. Furthermore, neither the explore network nor the exploit network were developed as a classification problem, while $\mathcal{D}$ is a classifier. In order to make these networks fit the GAN structure, there would need to be a restructuring of the CE architecture.

Another potential way to use the GAN is in replacement of the explore or exploit network. The explore network is the model that fits into the standard GAN formation. Indeed, it could be considered to be attempting a similar problem as $\mathcal{G}$. While there would be benefits of replacing the explore network with a GAN, limitations of the experiment setup prevent doing so. The amount of samples required to properly train a GAN would likely be prohibitive, considering how difficult it is to schedule time to conduct experiments on the space-based platform. As a result of this, the conclusion is that GANs are not directly applicable to the CE.

2.4 Summary

In this section, the framework was laid to establish the understanding of topics essential to this thesis. Section 2.1 described the foundations of machine learning on which the CE was developed. After this, the prior work established by the authors of [1] and [2] was detailed. The core issue of Catastrophic Forgetting was then explained, and potential mitigation techniques were looked at. Finally, GANs were explored, and ultimately deemed unlikely to work in the CE architecture. With the information described in this chapter, the reader is set to understand the work in the following chapters.

Chapter 3

Implementing Mitigation of Catastrophic Forgetting into Cognitive Space Communications Engine

In the following chapter, both software and hardware details of the test platforms used will be described. First, details about the software platforms will be included. This initially describes aspects of the software that are independent of the programming languages. Then, the language-dependent details will be provided. The CE was first implemented in MATLAB, to verify functionality. This was then ported to C++ using the MLPack library. Details about implementation in both languages will be described, both for the baseline implementation (CE-LM) and the two modified training methods (CE-RLM and CE-NSE). Once the software is described, the hardware setups for ground testing and flight testing will be described. Finally, the postprocessing will be briefly discussed.

3.1 Cognitive Engine Algorithm Details

The architecture of the CE follows the SARSA algorithm described in Algorithm 3. A flow diagram illustrating how data passes through the CE is shown in Figure 3.1. There are a couple differences between the CE and classic SARSA: the replacement of the Q-table with an MLP and the use of a separate MLP to guide the exploration of the network. The replacement of the Q-table with an MLP transforms the Q learning into a Deep-Q Network.

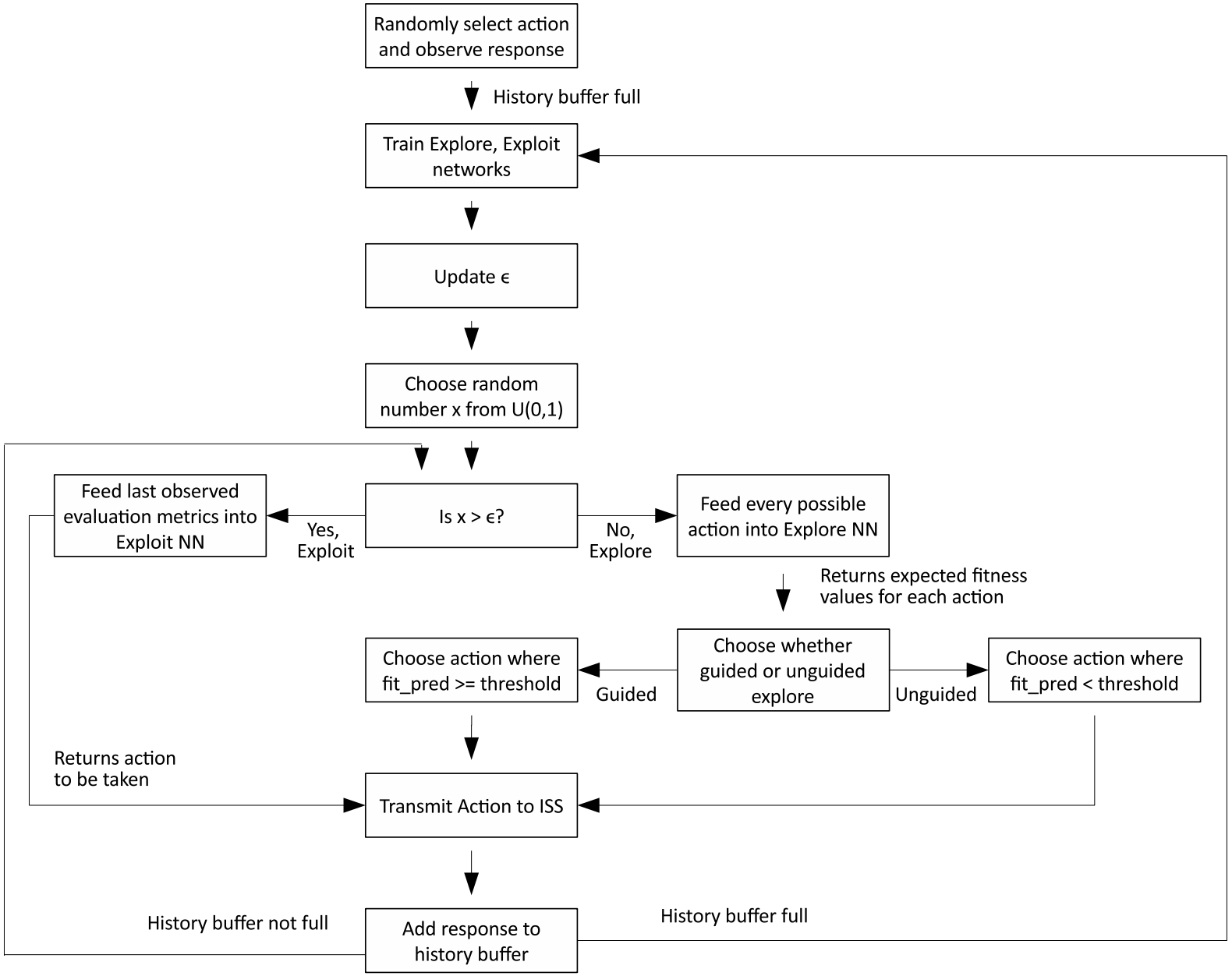


Figure 3.1: Flow diagram detailing the general logic of the CE.

The CE begins by randomly selecting actions and observing the results of these actions, storing unique action-reward pairs in a history buffer. In this experiment, the action is the specific configuration of PHY parameters, and the reward is a fitness score created by taking a weighted combination of different evaluation metrics. These metrics include thoughput, spectral efficiency, bandwidth used, power consumed on transmit, bit error rate (BER), and DC power consumed. Throughput, spectral efficiency, and bandwidth, DC power consumed and transmit power consumed are all calculated based on the chosen set of PHY parameters. A model of BER based on the other metrics is used as an estimate for BER that doesn't require more than one sample to confirm. Each parameter gets scaled from 0 to 1 based on the values that can be expected from each input. Then, they get combined in a weighted manner to produce the resulting fitness score. The importance of each metric is hard to determine, and can change depending on what priorities the system currently has. Six different weight combinations were arbitrarily selected to represent plausible use cases, and are shown in Table 3.1. Once the fitness score is calculated, the action-reward pair is stored in the history buffer until it fills up, after which the Explore and Exploit MLPs get trained. During initial training, the CE chooses an action that is fairly robust, to ensure a baseline result. In future training periods, there is no need to do this. Once done training, the CE has completed its startup process.

For the first step after training, the exploration threshold ϵ is set to 1, forcing an exploration. During each following step, the ϵ value is recalculated by taking the inverse of an incrementing counter. When this value passes a threshold, the counter is reset and exploration is again forced. After ϵ is updated, a number is drawn from uniformly random distribution $\mathcal{U}(0, 1)$. Then, if this number is less than ϵ, an explore iteration is triggered. Otherwise, an exploit iteration is triggered.

In the case of an exploration, all actions in the action space, as well as the

Table 3.1: Table containing different ways fitness score can be weighted. These weights were picked in [2] to be somewhat representative of possible priorities in space communication.

Mission Name	Throughput	BER	Target BW	Spectral Eff.	TX Eff.	DC Power Used
Emergency	0.1	0.8	0.025	0.025	0.025	0.025
Cooperation	0.05	0.05	0.4	0.4	0.05	0.05
Power Saving	0.05	0.05	0.05	0.05	0.3	0.5
Balanced	1/6	1/6	1/6	1/6	1/6	1/6
Launch	0.2	0.4	0.1	0.1	0.1	0.1
Multimedia	0.5	0.3	0.05	0.05	0.05	0.05

normalized observed SNR, are used as inputs to the Explore MLP block. The MLP returns a predicted fitness score for each action. From this, the actions are split up into two groups: actions that have a fitness score greater than the threshold and actions that have a fitness score less than the threshold. Then, with a 95% probability, an action is randomly selected from the group of actions with a greater fitness score than the threshold. The 5% probability of selecting from the other group of actions allows for exploration of areas that the Explore MLP block may have misinterpreted or areas that have been insufficiently explored.

In the case of exploitation, The normalized observed reward values of the last action are used as inputs to the exploit MLP block. Each member of the Exploit block has 6 component MLPs. Each component MLP corresponds to a normalized tuneable parameter. These six output values are then denormalized to get the actual output actions.

Regardless of if the action was chosen through exploration or exploitation, it is transmitted to the SDR platform (either in simulation or reality). The E_s/N_0 of the message is then received. This value, the energy of the symbol over the power spectral density of the noise, is a representation of SNR. Power consumed, power efficiency, bandwidth used, throughput, BER, and spectral efficiency are then

calculated, based on the action chosen and the E_s/N_0. Once these are all present, the evaluation metrics are normalized to a range of [0,1] and a fitness score is calculated depending on the mission that the CE is configured to run:

$$\begin{aligned} fitness =& w1 * Throughput_{norm} + w2 * BER_{norm} \\ &+ w3 * BW_{norm} + w4 * Spectral\ Efficiency_{norm} \\ &+ w5 * TX\ Efficiency_{norm} + Power\ Consumed_{norm} \end{aligned}$$

If the fitness value is greater than the maximum previously observed value, it becomes the new maximum observed value. If this value was achieved during exploration, the performance values are the next input to the exploit network. If the value is less than the maximum value and was chosen by exploitation, a new set of logic is followed. A value e_p is kept, representing the maximum fitness value while accounting for slow drift in fitness value responses resulting from slow-changing environmental properties. If the action taken was the same as the previous action, and the observed fitness is less than $0.9e_p$, then it is assumed the environment has undergone a rapid shift. The history buffer is reset and the system is reset to the initial state, where it randomly explores until the history buffer is filled. If the observed fitness is less than $0.9e_p$ but a different action was chosen, then the CE finds the best performing action from the history buffer and uses that action. Otherwise, if the observed fitness is greater than $0.9e_p$ and is not the first sample in the buffer, the new action is accepted. If the observed fitness is greater than $0.9e_p$ and no quick recover has occurred, the last known action is chosen as the new action to take. Finally, if the observed fitness is greater than e_p, e_p gets updated, and the last exploitation action gets chosen. This logic path is illustrated in Figure 3.2

Once the action-choosing logic has occurred, the observed fitness values are added to the history buffer. If the action chosen is unique to any action in the buffer, the

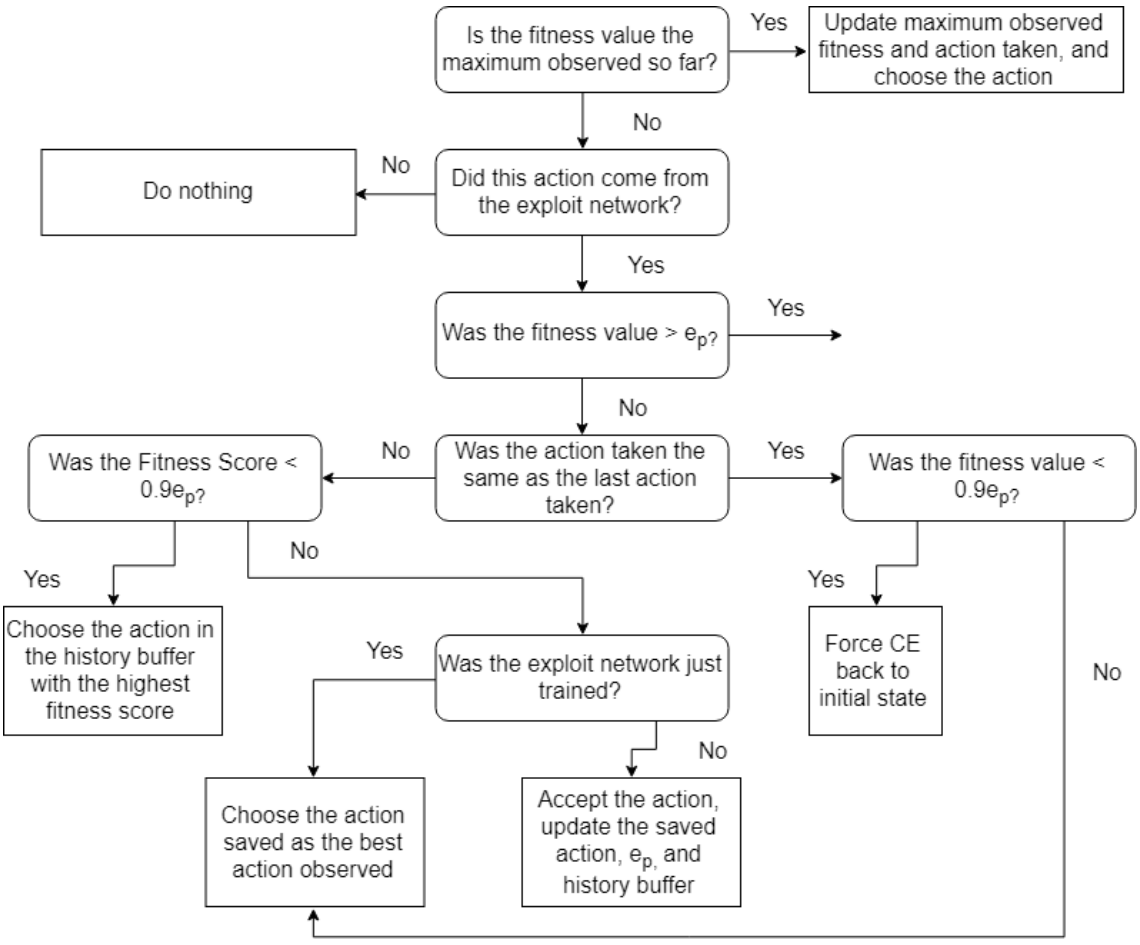


Figure 3.2: Flow diagram illustrating the logic process of filtering the action choice based on previously seen results. This logic allows the CE to catch if major changes should prompt a retraining period.

results are directly added to the history buffer. Otherwise, the CE updates the observed fitness value of the action in the buffer. If the history buffer is full, the Explore and Exploit networks get trained. In C++, the training process occurs in parallel with the rest of the CE so that the Explore and Exploit networks can continue to get used during training. The MATLAB simulation does not have the parallel training, but timing information from the simulation is not used. If the history buffer isn't full, the next iteration of the CE occurs.

Training occurs in two phases, one for Explore and one for Exploit. During each phase, data from the history buffer is randomly split up into training and

testing sets. Then, the training algorithm is applied, whether it is LM, RLM, or NSE. Details of the implementation of each will be described in the language-specfic sections.

The Explore and Exploit MLPs had different architectures, but maintained some similarities. Both used logarithmic activation functions, as well as a linear output function. The Explore MLP was composed of three layers. The input layer takes seven inputs, The first hidden layer has seven nodes, the second hidden layer had fifty nodes, and the output layer has one node. The Exploit MLP was actually six different MLPs, one for each parameter to be predicted. Each sub-MLP has two layers, with seven inputs, twenty nodes in the first hidden layer and one output.

In the past, each weight was initialized using MATLAB's default initialization method, Nguyen-Widrow initialization [67]. However, due to convergence issues the RLM method encountered while using this, the weight initialization was changed to the Glorot (Xavier) initialization technique [68], which sets weights by drawing from gaussian distribution $\mathcal{N}(0, \sigma^2)$, where:

$$\sigma^2 = \frac{2}{n_{in} + n_{out}},$$

and n_{in} and n_{out} are the number of input and output nodes in the network.

3.2 Software Methods

This section covers the specific software implementations of the CE, first in MATLAB then in C++.

3.2.1 MATLAB Simulation

Prior to this thesis, the CE was prototyped in MATLAB by Paulo Ferriera [2]. It was developed in MATLAB 2015a, using the Parallel Computing Toolbox and the Neural

Network Toolbox. It is important to note that the Statistics and Machine Learning toolbox was used and not the Deep Learning toolbox. Both toolboxes are able to create the CE, but have different interfaces. The following section will describe in summary the structure of the baseline simulation, as well as the modifications required to implement CE-RLM and CE-NSE. The full MATLAB code base can be found in Appendix C.

For all learning methods, the MATLAB neural network structure was used as the foundation. However, for CE-RLM and CE-NSE, the framework was augmented with training functions specific to each algorithm. Most of the previous implementation of the CE was untouched during the thesis. Indeed, the only part that had major changes is the training function, which is abstracted away by default.

By the nature of RLM, there are a handful of parameters and intermediate values that need to be kept track of. This was done using a MATLAB struct, RecurseMatrix, found in Appendix C.7. It contains the current Gradient, time, P matrix, ρ, S, α, and the size of a batch to be trained. Beyond that, the algorithm described in Section 2.7 is followed in a straightforward way.

The implementation of Learn++.NSE was modified from the version provided by [69]. When working with the US government, International Traffic in Arms Regulations (ITAR) constraints must always be considered. The code provided by [69] uses a GNU General Public License (GPL), which requires that the source of a program using the library is accessible under the same license. This poses issues to ITAR-controlled code. However, the MATLAB simulation has no ITAR sensitive material, so the license did not matter in this case.

The main changes to the code provided by [69] involved adjusting the structure to fit a regression problem instead of a classification problem. Adaboost.R1 was used as a reference point in doing this adjustment, as it is similar in operation, if not how it weighs samples [55]. The cross-entropy loss function that the standard

Learn++.NSE algorithm used [57] was replaced by mean squared error (MSE). In addition, during prediction the result was a weighted average of the ensembles, instead of a weighted majority vote. Other than the changes necessary for adapting Learn++.NSE to regression, not much was changed, beyond a handful of extra hyperparameters.

3.2.2 C++

The software architecture used in ground and flight testing was initially introduced by the authors in [1]. In order to ensure generality, performance, and reusability, an Object-Oriented programming language was most suitable. C++ was chosen, both for performance and for some of the constructs in the C++11 standard. In addition, many helpful open-source C++ libraries exist. The software was designed and compiled in an Ubuntu 16.04 Linux VM, but was designed to be recompilable for any x86-based operating system, and can, with work, be used with most processor types that have a C++ compiler.

In the design of the CE, drivers for the multiple modems needed to be created. These drivers are restricted by ITAR, ruling out the use of many libraries with "copyleft" licenses, such as the GNU GPL. This greatly reduces the pool of possible libraries to use.

Despite the wide variety of publicly available ML libraries, many of them do not have APIs for lower-level languages such as C/C++. While this is rapidly changing with the development of embedded ML applications, at the time of the initial CE construction the selection of C++-compatible libraries was small [1]. Furthermore, many of the existing C/C++ libraries lacked in-depth documentation for the lower-level APIs, had very complex build processes, or were built for larger, more complex applications (such as deep, convolutional NNs).

MLPack [70] was the library chosen to implement the MLPs in the CE. Because

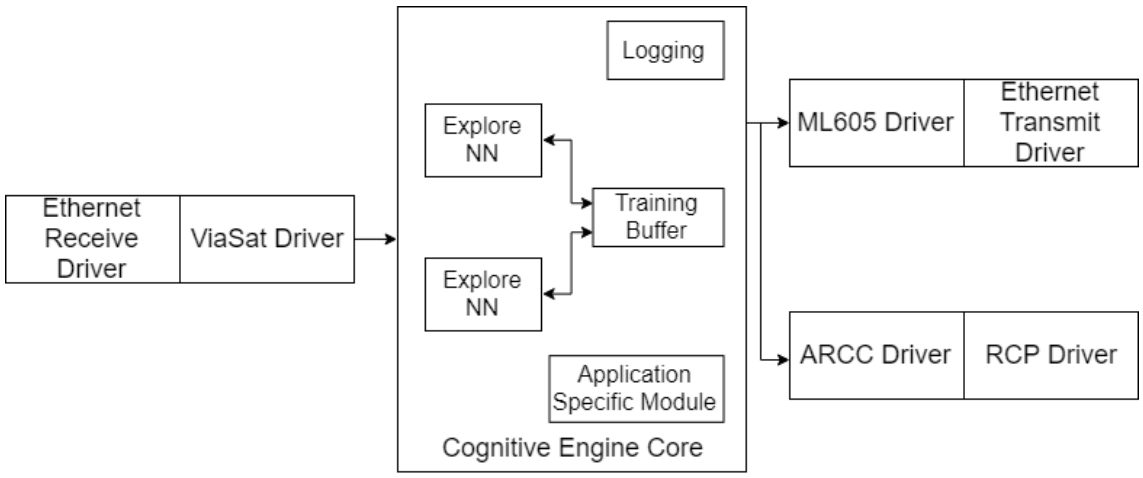


Figure 3.3: An outline of the Cognitive Engine software architecture. MLPack is used primarily in the Explore and Exploit NN objects, and Boost.ASIO was used in Ethernet libraries. All other libraries were used throughout the software architecture.

of the time of development the version used was 2.1.1. Since then, MLPack has revamped its ANN architecture, so the library version is important when attempting to replicate these results. MLPack is built on top of Armadillo 7.600.2 [71], so Armadillo was used for self-written matrix/vector operations and storage as well. By doing this, the interfacing between self-written components and MLPack was simple.

In order to simplify the interfacing that the CE has to do with Ethernet and UDP/IP for communication with modems (described in Section 3.3), the Boost.ASIO [72] library was used. This library abstracts away any operating system-specific constructs for handling sockets. Finally, the Boost.Serialization [72] library was chosen for saving/resuming CE states.

A block diagram of the CE is shown in Figure 3.3. During the ground tests, the external drivers used are the ViaSat driver and the Advanced Radio for Cognitive Communications (ARCC) driver. The ViaSat modem is used to get the observed Es/No value and the ARCC to simulate the response of the SCaN testbed from the action chosen by the CE. Since the ViaSat modem is connected through ethernet, an

ethernet driver is necessary as well. During flight tests, the ARCC driver is replaced by the ML-605 BPSK driver, which also interfaces with the ethernet driver. The CE being tested resides in the RLNN Core section. The description of the RLNN Core will be split into three parts: the baseline method used in 2017, the changes required for the implementation of RLM, and the changes required for the implementation of Learn++.NSE.

The Explore and Exploit MLP ensembles were grouped into an NN Predictor object, allowing for references to the MLPack library to be abstracted out. In this implementation, any ensembles used were different solely on initialization of the weights. As such, both the Explore ensemble and Exploit ensembles had an effective ensemble size of 1. Within the NN Predictor object is the FeedForwardNetwork object, which implements the individual FFNN (MLP) and the implementation of Levenberg-Marquardt written by Timothy Hackett in [1]. As described in 2.3.1, the CE uses online training. This required two NN Predictor models for each member of the ensemble, one to get trained and one for execution of the model. The two-NN Predictor structure allows for concurrent training and usage, only halting when weights are being copied over. Training occurs using one NN Predictor model, and then the weights get copied to the other NN Predictor model.

The buffer of training data, which is common between Explore and Exploit NN Predictors, is implemented in a separate object. It holds the latest 200 unique actions. When training occurs, the buffer is split into training and validation sets, and is transformed for each type of MLP (as Explore and Exploit MLPs have different inputs and outputs). The Armadillo library is used to perform this splitting and transformation.

For the most part, the RLNN core of the CE is kept generic so that it can be applied to problems outside of communications. The Application Specific Module provides the satellite communications context for the RLNN core. It transforms

communication metrics into the fitness scores that the rest of the RLNN Core uses. Any patches that change system behavior occur in this module as well. One example of this is the patch that limits transmit power changes to 1.5 dB steps. Another example is that the fitness scores are zeroed out when BER was measured to be 0.5, as this implies that the system had no communication between transmitter and receiver.

The key differences between CE-RLM and CE-LM are located in the FeedForwardNetwork object. Here, the LM function was replaced by a function that runs RLM instead. Because of the additional parameters that RLM uses, a separate class called RecursiveLMHelper was used to contain the parameters and abstract the matrix math in the RLM updating process. In addition to these changes, CE-RLM used an ensemble of size 2 for the Exploit MLP structure. This was done to mitigate some of the instability of RLM. The Explore ensemble was kept at size 1, as each additional ensemble greatly increases execution time of the Explore MLP.

Unlike RLM, Learn++.NSE was mostly implemented as a modified version of NeuralNetwork Predictor, renamed LearnNSEPredictor. This is because the foundational MLP aspects of training are the same as the CE-LM. Where the difference occurs is the weighting of the samples that are being trained on and how the predictor is used. As such, the main changes occur in the train and predict functions. The Learn++.NSE algorithm is described in 4, and so will not be reiterated here. CE-NSE used a somewhat arbitrarily chosen Exploit ensemble of size 8. For similar reasons to CE-RLM, CE-NSE kept the Explot ensemble to size 1.

Post-processing the data gathered from the execution of the C++ implementations of the CE proved to be slightly more complicated than the MATLAB simulation, which had all relevant data easily exposed after running. The C++ CE was configured to keep a human-readable text log detailing the configuration of the CE. It also logged details about each iteration through the CE, including the start time,

whether exploit or explore was chosen, the time that the action was chosen, the action chosen, the time an response was measured, the evaluation metrics received, the fitness score observed, and whether the network was training. The authors of [1] provided a MATLAB script to parse the log files, which is included in Appendix C.2. This parser generated figures that illustrate the results of a pass a time series as well as the amount of time it took to run the CE. It also generated MATLAB workspaces that contained useful information from the logs, which were used in the rest of the post-processing that occurred.

3.3 Hardware Methods

In the following section, the hardware used in the ground testing and the on-orbit testing.

3.3.1 Ground Test Setup

During the ground tests that were conducted during July of 2018, the test setup was very similar to that used in 2017 [1]. Similarly to that work, a testbed at NASA GRC was used to create an emulation of the expected on-orbit environment. A simplified block diagram of the setup is shown in Fig. 3.4. The CE is contained in an Ubuntu 16.04 virtual machine on a Windows 7-based Dell T3600 Precision workstation. The VM was allocated 4 GB of dedicated memory, and shared all eight of the CPU cores (with hyper threading) with the host operating system. The CE is put on a private local area network with the ARCC test platform and the ViaSat DVB-S2 receiver. The CE transmits an action to the ARCC, which then transmitted DVB-S2 frames through a channel emulated by a variable attenuator, according to a measured SNR profile from a previous on-orbit experiment. The noise floor can be adjusted by the noise generator. All RF signals in this test were passed using coaxial cables, insetad

of any antennas.

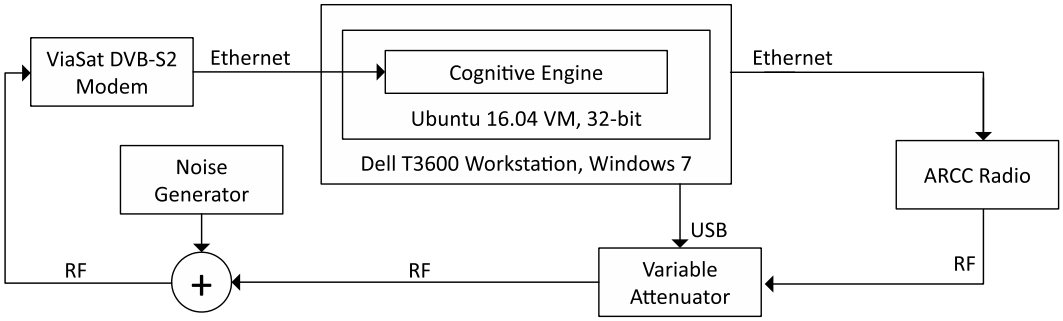


Figure 3.4: Block diagram illustrating how data flows through the hardware setup used for the ground test.

3.3.2 Flight Setup

The setup for on-orbit testing was identical to the setup used in 2017 [1], which in turn used a setup initially used by previous NASA GRC collaborators. A simplified block diagram of the setup is shown in Fig. 3.5. In the chain, there are two DVB-S2 RX modems. The ViaSat modem was used for sending E_s/N_0 measurements at a rate of 100 Hz over UDP. The Newtec modem demodulates and decodes the actual frames coming in, and saves it to a binary file for postprocessing. During each dataframe, the CE saves the previous action tuple along with its performance. It then chooses the next action tuple, and sends the next action to the ML-605 BPSK modem, which then is used to uplink the chosen action to the SCaN Testbed.

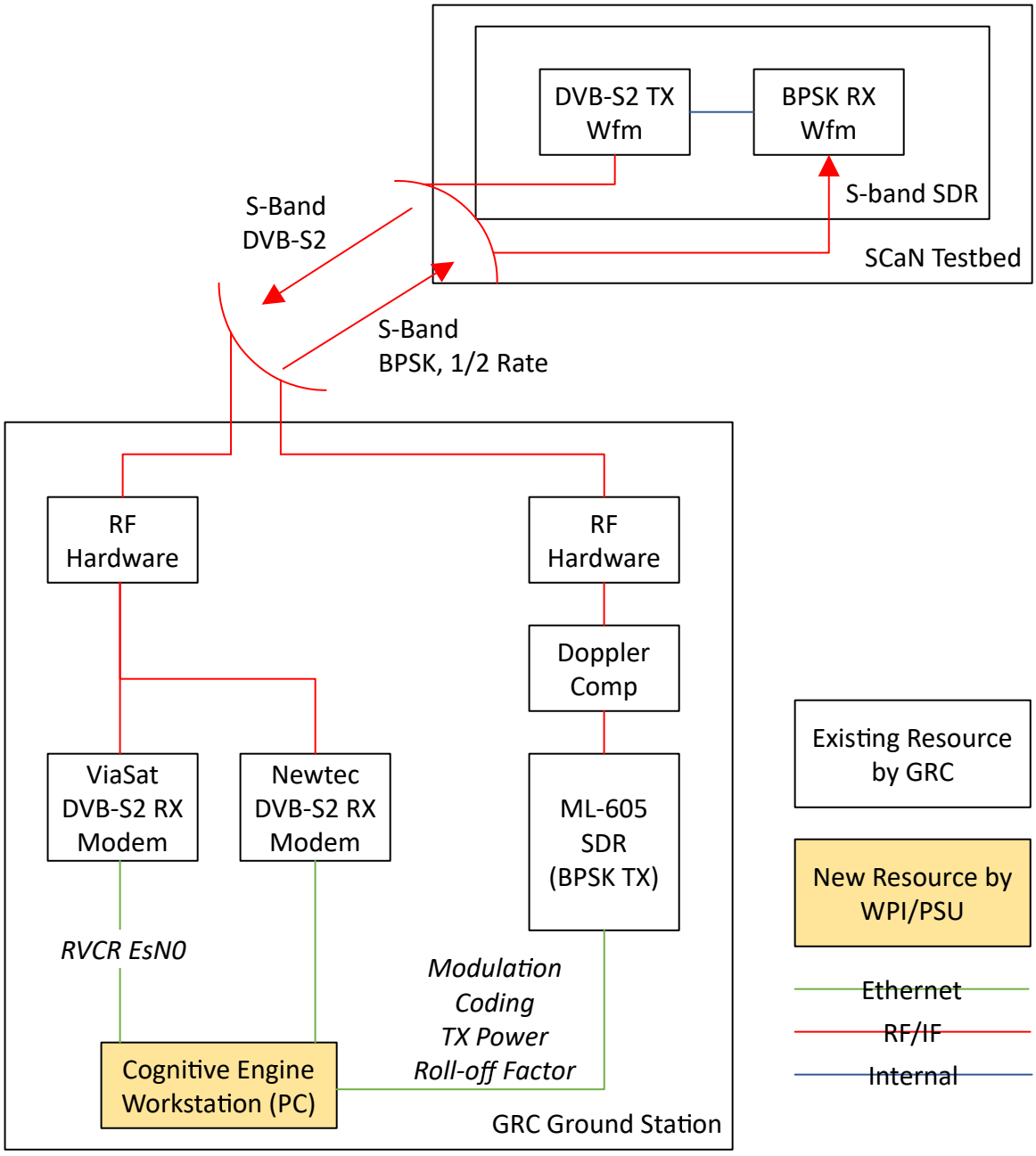


Figure 3.5: Block diagram illustrating how data flows through the hardware setup used for the flight test.

3.4 Summary

In this chapter, details about incorporating methods to mitigate catastrophic forgetting into the CE were described, both for the MATLAB simulation and the C++ implementation to be used for flight testing. Doing so first required discussion of the baseline CE-LM, before details about CE-RLM and CE-NSE could be described. The hardware setups used for ground tests and flight tests were also included.

Chapter 4

Performance Results of Catastrophic Forgetting Mitigation Techniques

In the following chapter, the results of the implemented Catastrophic Forgetting mitigation techniques will be explored. There are three main categories of results collected: MATLAB simulation results, C++ simulation results, and C++ flight test results.

4.1 MATLAB Simulation Results

MATLAB was initially used in developing the modified CEs, both with the RLM training method (CE-RLM) and the Learn++.NSE training method (CE-NSE). The intent of the MATLAB simulation was to verify the potential benefit of both changes to the CE, in an environment where implementing iterational modifications is as frictionless as possible. More realistic analysis, including timing analysis, was conducted in the C++ CE implementation. As such, the SNR profile used in MATLAB-based testing was a simple slow-fading channel, so that degenerate be-

havior would be definitively a result of poorly implemented code. A plot of the SNR profile is shown in Figure 4.1.

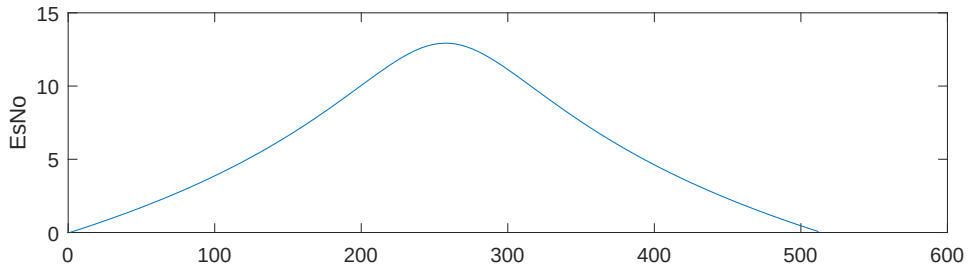
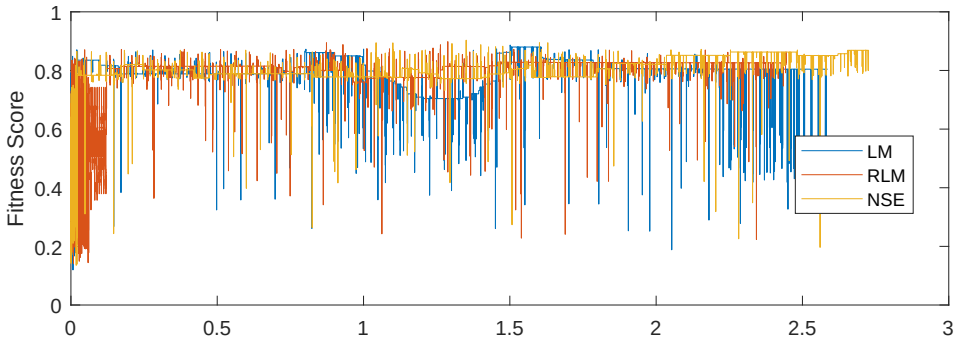


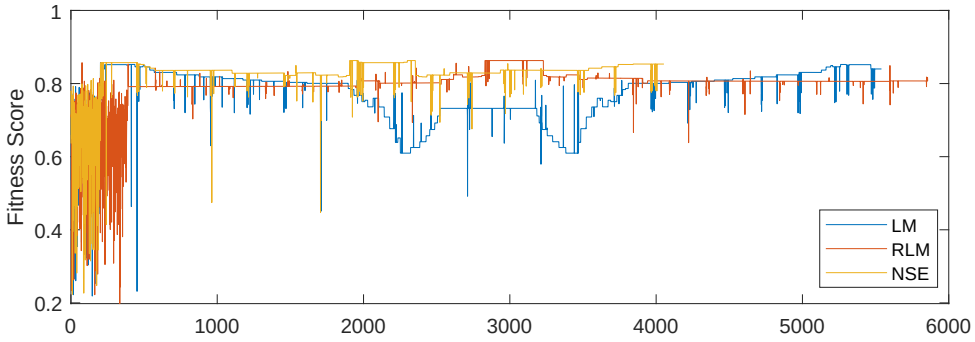
Figure 4.1: SNR profile used in MATLAB simulation.

While there are six different fitness score weightings (as shown in Table 3.1), the flight tests conducted in [1] focused on Emergency, Cooperation and Power Saving. Because of this, these were the missions that the MATLAB simulation focused on as well. Figure 4.2 shows how the fitness score evolved over time during the simulation.

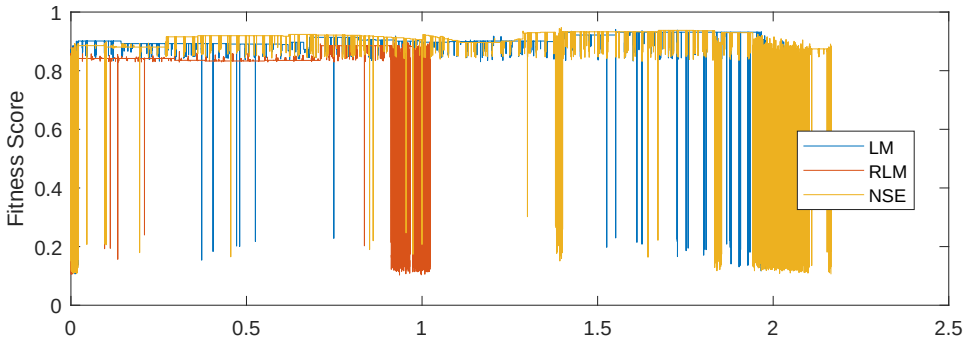
A few things are worth bringing up with the time series plots shown in Figure4.2. The first is that the spurs that are periodically showing up are the CE exploring random actions, and are spaced in a manner independent of training type. Instead, the spacing is solely dependent on the value of the exploration parameter ϵ. In addition, it is apparent that some mission types have different ranges of fitness scores that are possible within the action space, as the Cooperation mission simulation encounters a wider range of values during exploration than either the Emergency mission simulation or the Power Saving mission simulation. Beyond this, it is evident that LM has a more difficult time than the other two training algorithms in adapting to the higher SNR portions of the simulated pass in both the Cooperation and Power Saving mission cases. Other than this, it's fairly difficult to draw conclusions from these time series plots.



(a) Fitness score plotted over length of simulation, Cooperation mission.



(b) Fitness score plotted over length of simulation, Power Saving mission.



(c) Fitness score plotted over length of simulation, Emergency mission.

Figure 4.2: Fitness score plotted over length of simulation.

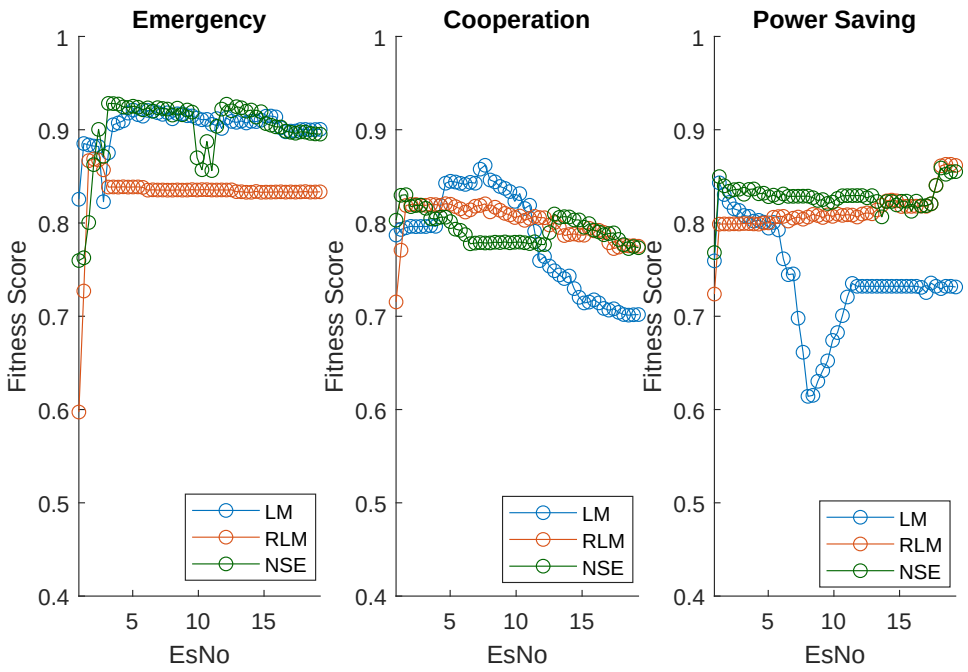


Figure 4.3: Binned means, MATLAB simulation

In order to get a better understanding of the behavior of the different modifications of the CE, fitness scores were split into bins based on the E_s/N_0 at the moment that the fitness score was observed. Once this was done, the mean was taken within each bin. This plot is shown in Figure 4.3. For the Emergency mission, CE-LM and CE-NSE proved to have similar mean fitness scores, with CE-RLM having lower mean fitness scores. The Cooperation mission was more ambiguous, with CE-RLM and CE-NSE performing better than CE-LM in the higher E_s/N_0 regime, and performing worse than LM in the middle E_s/N_0 regime. Finally, the Power Saving mission shows both CE-LM and CE-RLM working markedly better than CE-LM in the entire regime.

With the results shown in this section, there was enough motivation to progress from the MATLAB simulation to simulation, ground testing, and flight testing in C++.

4.2 C++ Simulation Results

Simulations using the C++ implementation of the CE served two main purposes. The first purpose of C++ simulations was to verify the correct operation of the code prior to connecting the CE to any hardware. The second purpose is to provide a completely identical E_s/N_0 profile with which to compare the different training methods. As much as the NASA flight staff intends to be consistent when evaluating the quality of the communications channel during windows when the ISS is in line of sight, the variation of conditions within each quality category resulted in the different tests conducted between 2017 and 2018 having pass qualities that were hard to directly compare. Running a simulation allows for more direct comparison, albeit without the possibility of shifting the action space in a way that breaks the communication link, preventing an important failure case.

The SNR profiles used in the C++ simulation come from attenuation profiles that were measured by NASA employees at GRC. These profiles are used with variable attenuators to simulate channel conditions. By subtracting values in these profiles from the maximum E_s/N_0 that the CE is expected to see, the attenuation is transformed into suitable E_s/N_0 values. These values were fed to the CE. One example snrProfile is shown in Figure 4.4, while the rest can be found in Appendix B.1.

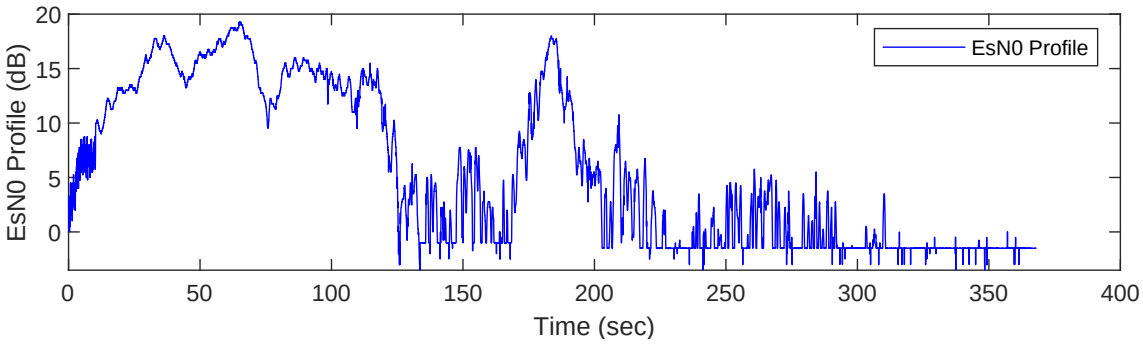


Figure 4.4: One of the SNR profiles used in C++ simulation.

The E_s/N_0 profiles for the C++ simulation are significantly more complex than

the E_s/N_0 profile used in the MATLAB simulation. This is because the slow fading channel simulated in MATLAB is oversimplified for the case of satellite communications. One of the key differences between CE-LM and the new methods being studied is that both CE-RLM and CE-NSE can get added value from pretraining. RLM can explicitly learn a better representation of the environment, while Learn++.NSE can build up a cache of pretrained networks. LM, on the other hand, will replace the information learned in pretraining upon the first retraining period, and so CE-LM did not utilize pretraining.

Running the C++ CE implementation generates a logging file, containing the salient information from each iteration of the algorithm, including the time it takes to choose an action, the fitness score observed, and the other evaluation metrics that combine to become the fitness score. Figure 4.5 is a summary plot of CE-LM running LM operating the Cooperation mission on the SNR profile shown in Figure 4.4. The same plots for CE-RLM and CE-NSE are shown in Figures 4.6 and 4.7 respectively. The key difference that's noticable from this series of plots is that RLM takes a significantly longer amount of time to train, as expected. The plots also provide some intuition about what the action space is actually representing. Even with this, this set of visualizations is hard to utilize for comparing training methods.

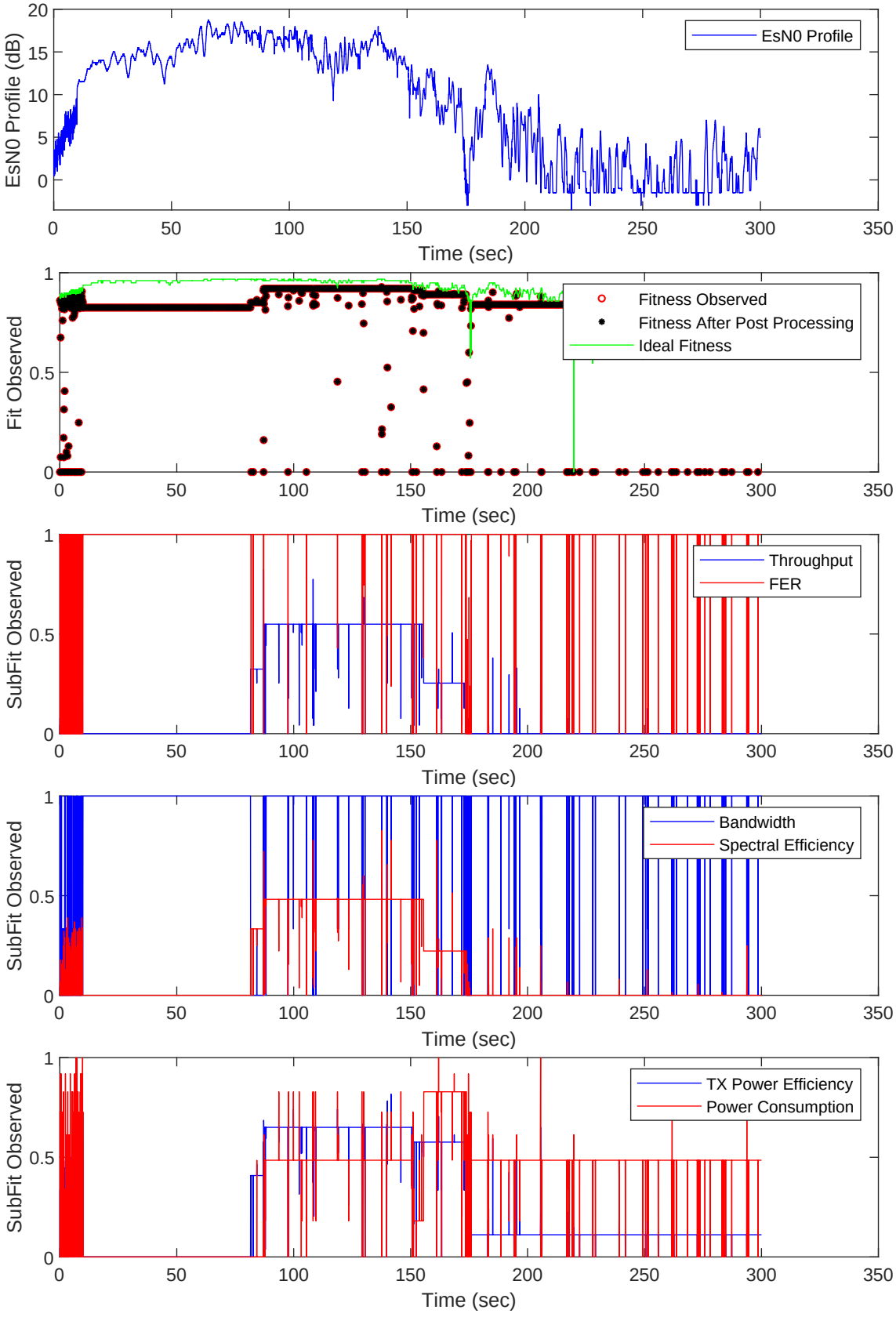


Figure 4.5: Operation of CE-LM on SNR profile 22, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

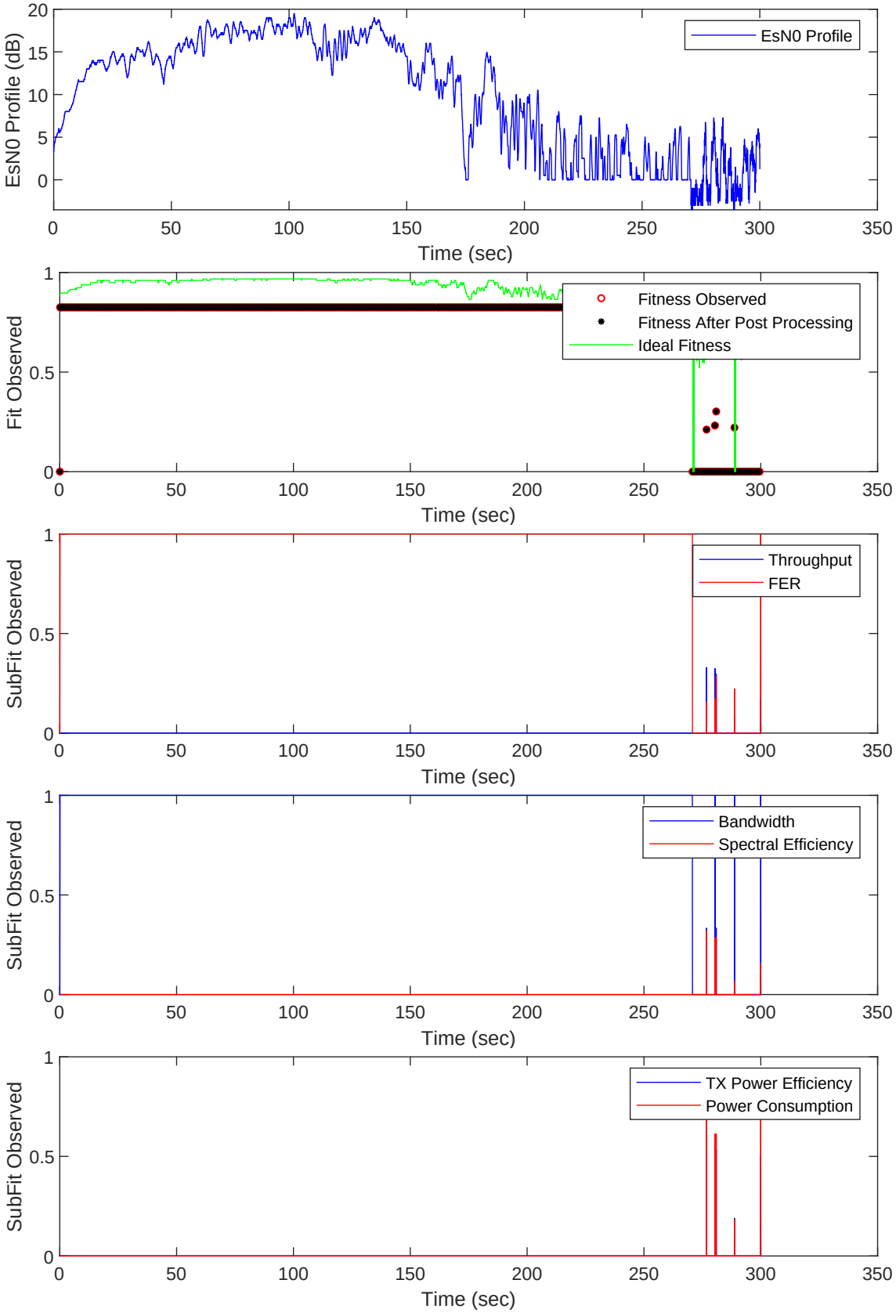


Figure 4.6: Operation of CE-RLM on SNR profile 22, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

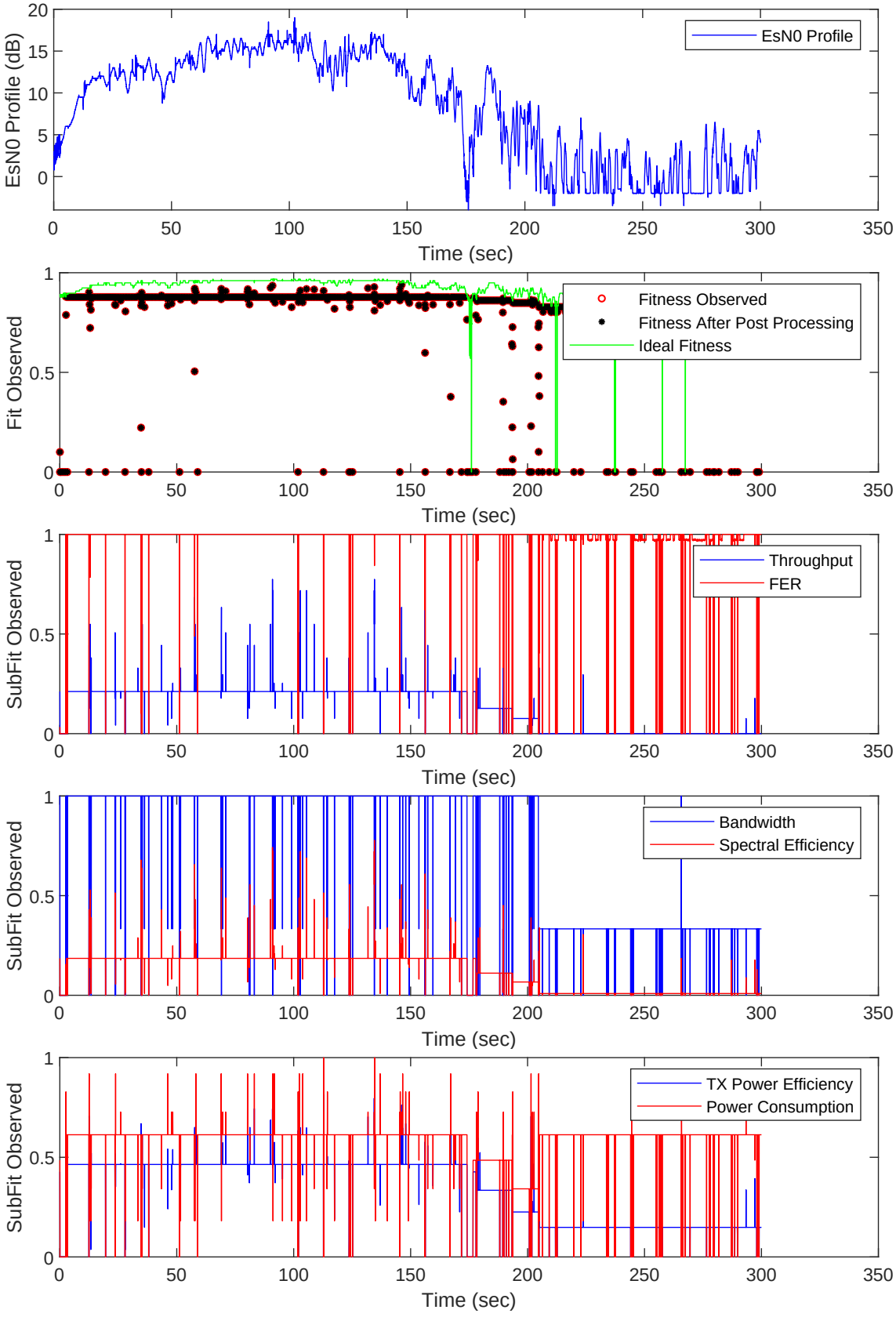
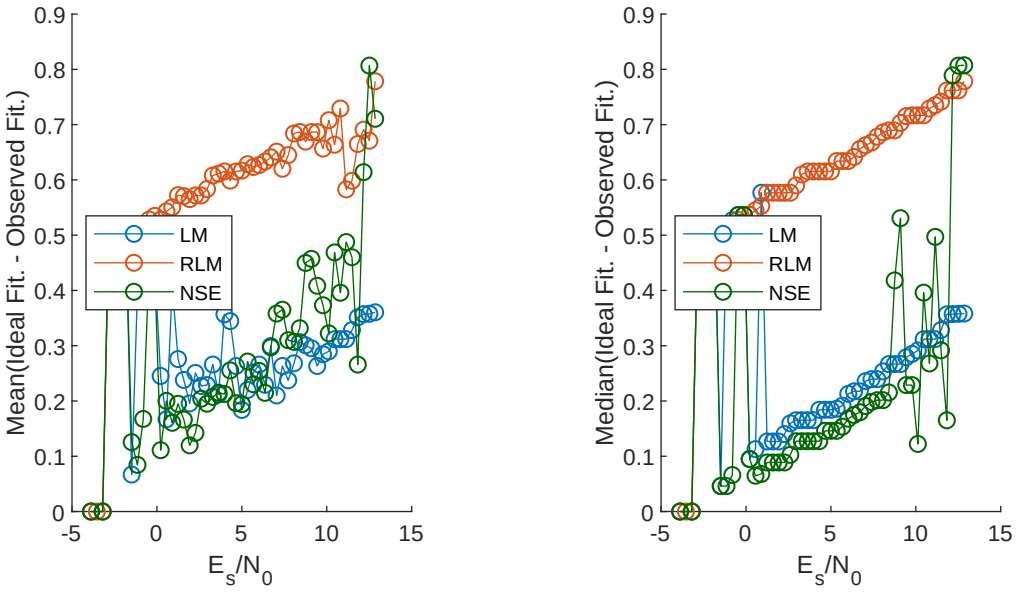


Figure 4.7: Operation of CE-NSE on SNR profile 22, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

Like the MATLAB simulation, the fitness scores were binned by E_s/N_0, and the mean and median were taken within each bin. Unlike the MATLAB simulation, however, the postprocessing of the CE log files includes the calculation of the ideal action to be taken during each iteration, given the last observed state. This was left out of the MATLAB simulation because the training methods use the same SNR profile, and could be compared more directly to each other than results from the flight test. Because the calculation of the ideal action was already built into the postprocessing script, the C++ simulations utilize this ideal action and fitness score in evaluation. Using the ideal fitness score acheivable, the fitness values observed are normalized by subtracting them from the ideal fitness value. This will be referred to as the fitness distance and will replace the raw fitness scores for the majority of the following plots.



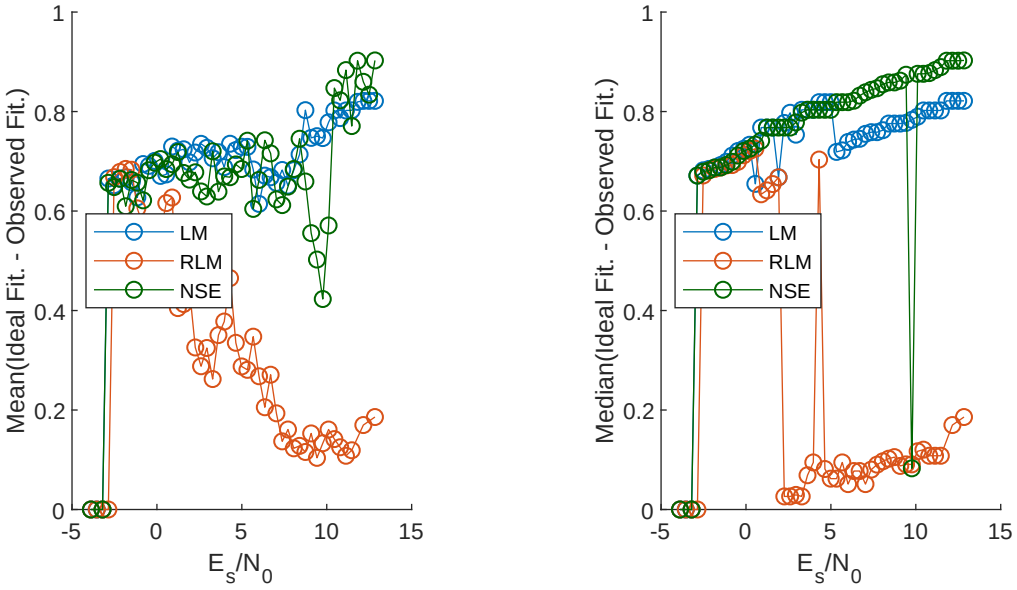
(a) Binned mean of fitness distance between ideal fitness and observed fitness.

(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure 4.8: Binned mean and binned median plots for the different CE training methods running the Cooperation mission. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores, then getting the mean or median within each bin.

The binned mean and median plots are shown in Figure 4.8. The first notable

observation is that CE-RLM performed significantly worse than either CE-LM or CE-NSE. This behavior may be attributed to the RLM algorithm requiring more training iterations to properly converge vs the other algorithms, which combined with the extended time it takes to train is resulting in non-convergence. This non-convergence seems to be situational to individual pass/performance combinations. For instance, running the Power Saving mission on the same E_s/N_0 profile results in CE-RLM outperforming CE-NSE and CE-LM. This is shown in Figure 4.9. In this case, CE-RLM likely converged.



(a) Binned mean of fitness distance between ideal fitness and observed fitness.

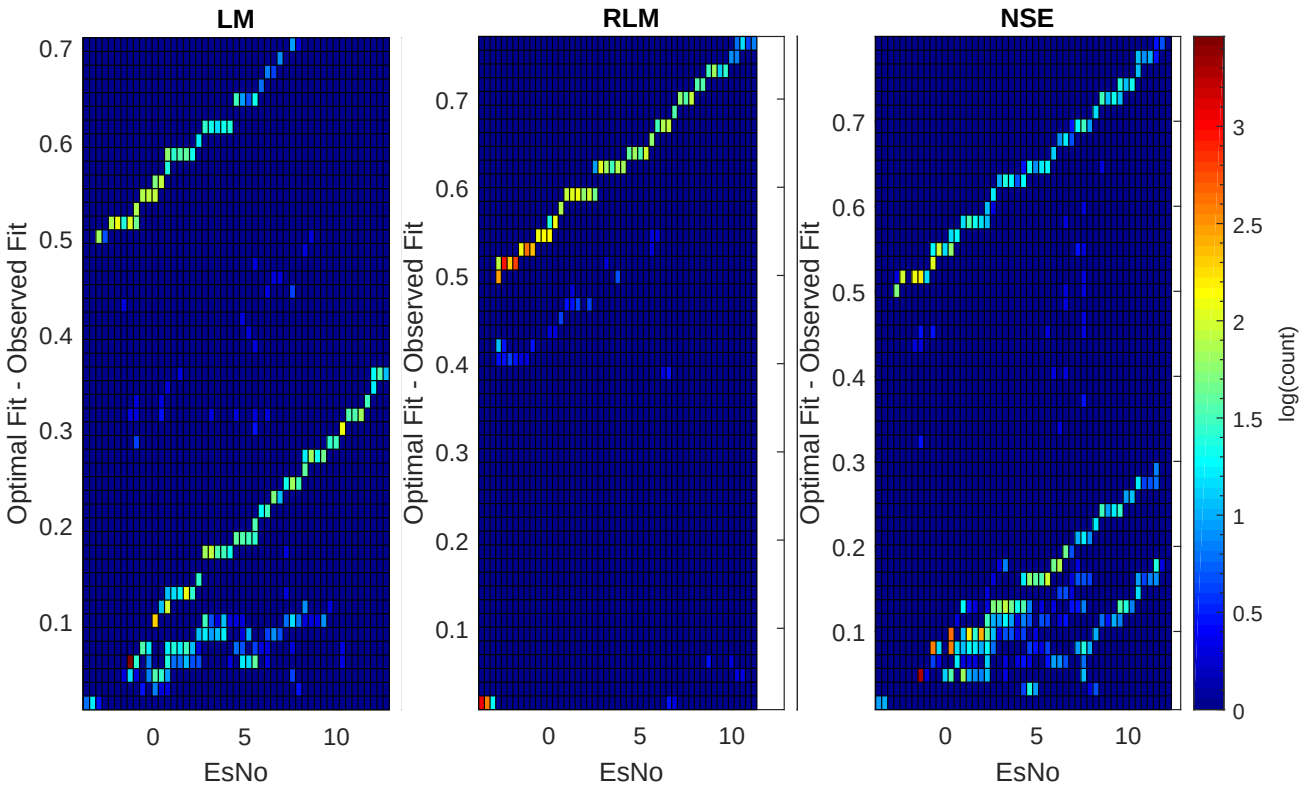
(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure 4.9: Binned mean and binned median plots for the different CE training methods running the Power Saving mission. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores, then getting the mean or median within each bin.

The binned mean and median representations allow for a simple evaluation of performance between the training methods, but is potentially an oversimplification, collapsing multiple CE behaviors into one number. A different representation of the data is shown in Figure 4.10. In this figure, a 2-dimensional histogram is taken. One dimension is the E_s/N_0 bins, in the same way as the binned means plot previously.

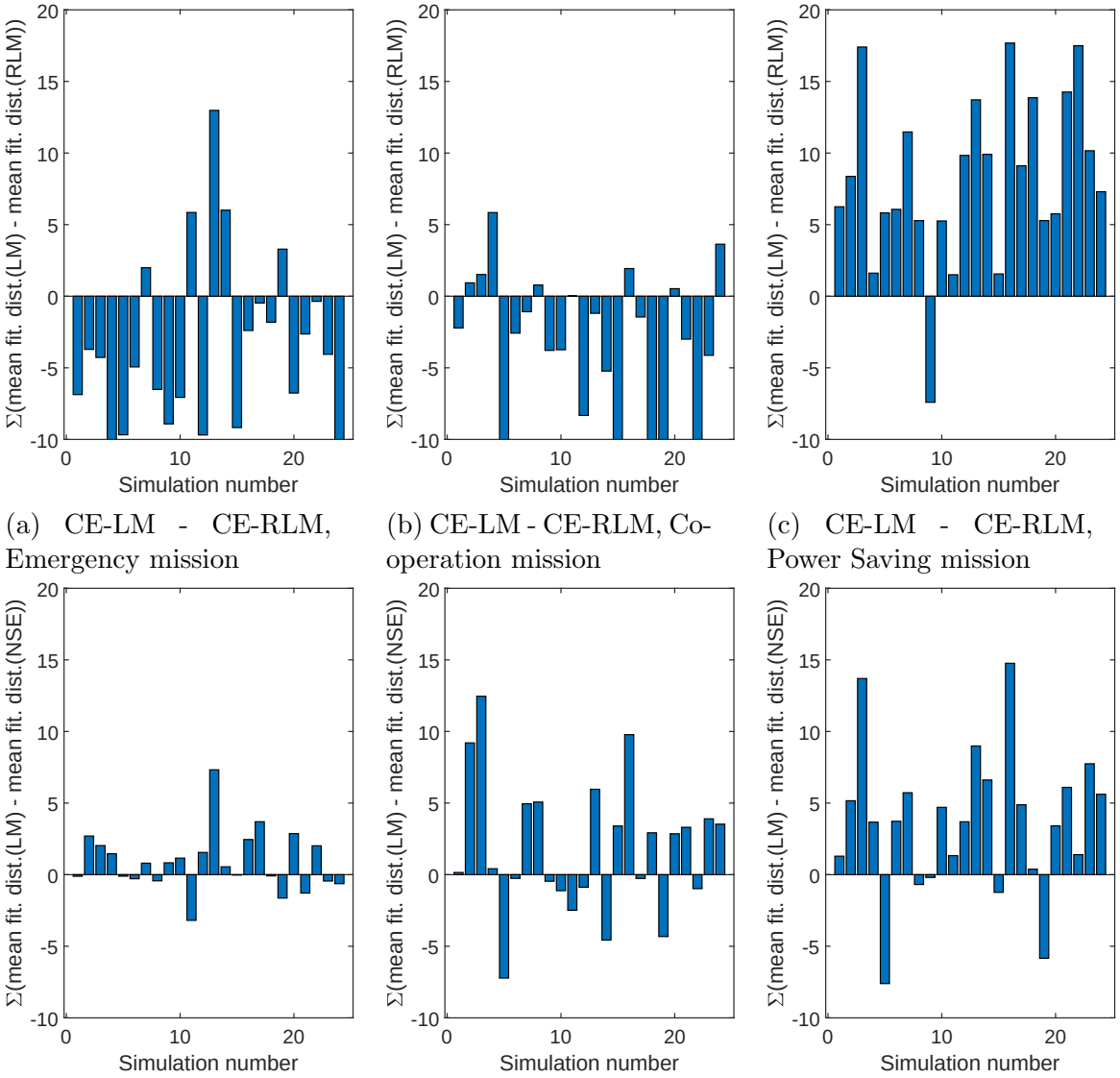
The other dimension is the fitness distance observed. The counts are then put on a log scale. This style of plot makes it easier to understand what the CE is actually doing. Each visible line represents an action or cluster of actions selected by the CE. At lower E_s/N_0 values, these actions often approach the best action that can be taken. When the CE has identified an action that functions well, it tends to not explore far enough to locate one that may work better. This results in the same action continuing to be chosen for higher E_s/N_0 values, where it's performance may not be as close to the ideal one, despite being close to ideal at lower E_s/N_0 values. The consistent bar of high fitness distance actions in the plots represent the failure case of the CE, when the action chosen reports zero fitness. The repeated zero fitness score iterations potentially skew the mean of any binned mean values. Binned Medians are also included, in order to reduce the impact of the zero-based skew. More of these plots are shown in Appendix B.2.

In order to approach a single-number evaluation of the performance of the different training methods, the binned mean fitness distances for each method is compared. For each SNR profile used, the difference is taken between the fitness distances of CE-LM and CE-NSE, and CE-LM and CE-RLM. This difference turns out positive if the first method in the difference has a higher fitness distance than the second method, implying the second method performed better. Likewise, this number is negative if the second method has a higher fitness distance than the first method. The difference in means is then summed up. This is done for each of the simulations, and is shown in Figure 4.11.



(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure 4.10: Two-dimensional histograms of each training method operating the Cooperation mission. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



(a) CE-LM - CE-RLM, Emergency mission

(b) CE-LM - CE-RLM, Co-operation mission

(c) CE-LM - CE-RLM, Power Saving mission

(d) CE-LM - CE-NSE, Emergency mission

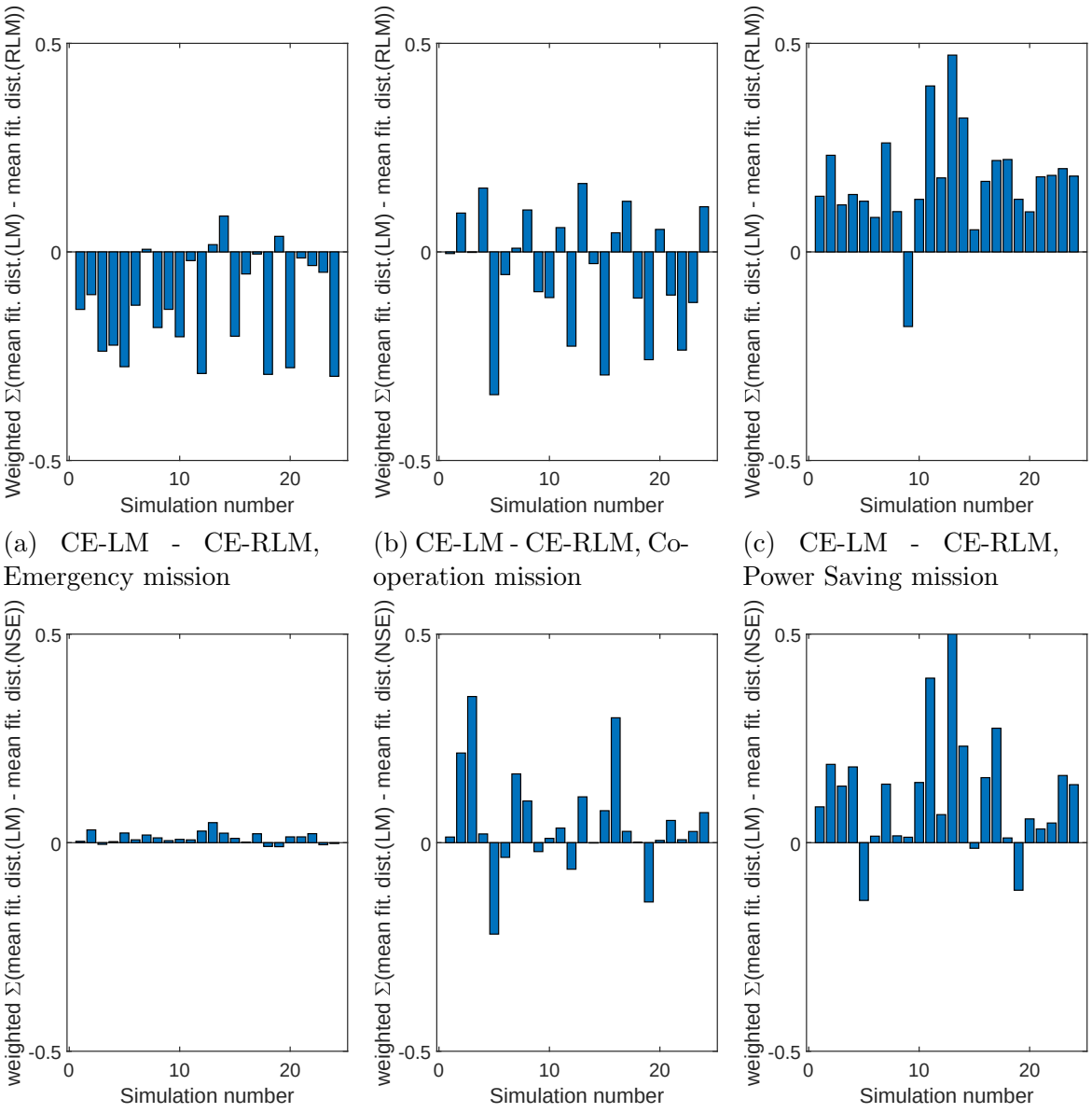
(e) CE-LM - CE-NSE, Co-operation mission

(f) CE-LM - CE-NSE, Power Saving mission

Figure 4.11: A series of bar plots taking unweighted sum of the difference between binned means from CE-LM and either CE-RLM or CE-NSE. (a),(b), and (c) are between CE-LM and CE-RLM, while (d),(e), and (f) are between CE-LM and CE-NSE.

For the Emergency and Cooperation missions, It is fairly clear that CE-NSE outperforms CE-RLM, with the majority of the fitness distance sums being positive. Looking at Figures 4.6 and 4.7, this appears likely due to the fact that the exploit networks having not properly converged to an action given the E_s/N_0 behavior. However, as mentioned before this isn't always the case, as CE-RLM outperforms both CE-LM and CE-NSE for most SNR profiles during the Power Saving mission. For all missions, CE-NSE appears to have modest improvements on the performance of CE-LM overall. The moderate but not overwhelming improvements can be attributed to the fact that the base learning algorithm is the same, and all the improvements come from clever usage of ensembles.

One potential flaw of the plots in Figure 4.11 is that a bin that has a small number samples in it would have the same impact on the plot as a bin that has a large samples in it. In order to mitigate this, the same plots were recalculated, except the mean and median values of the bins were multiplied by the number of samples in the bin divided by the total number of samples. This way, a good performance in a common E_s/N_0 regime won't be overshadowed by a poor performance in an uncommon E_s/N_0 regime. These modified plots are shown in Figure 4.12. This normalization changed some of the results for individual SNR profiles, but the general patterns observed remain the same.



(a) CE-LM - CE-RLM, Emergency mission

(b) CE-LM - CE-RLM, Co-operation mission

(c) CE-LM - CE-RLM, Power Saving mission

(d) CE-LM - CE-NSE, Emergency mission

(e) CE-LM - CE-NSE, Co-operation mission

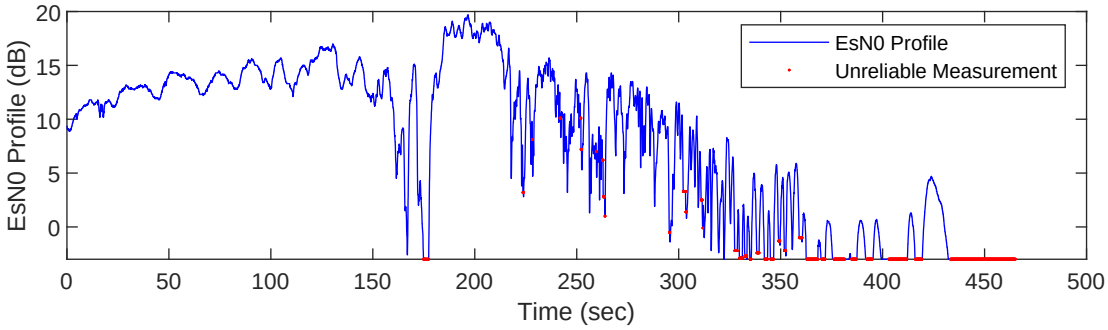
(f) CE-LM - CE-NSE, Power Saving mission

Figure 4.12: A series of bar plots taking a weighted sum of the difference between binned means from CE-LM and either CE-RLM or CE-NSE. (a),(b), and (c) are between CE-LM and CE-RLM, while (d)(,e), and (f) are between CE-LM and CE-NSE.

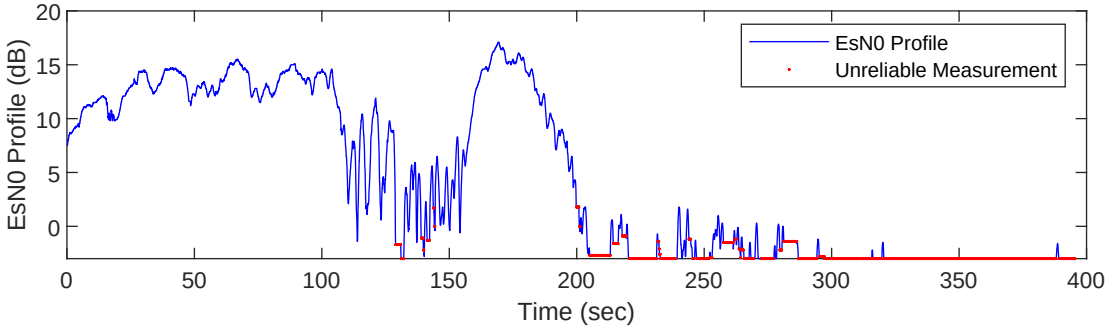
4.3 Real-World Flight Test Results

Flight tests were conducted in two different time periods: the testing of CE-LM were conducted between May 2, 2017 and May 12, 2017, as described in [1], while the testing of CE-RLM and CE-NSE took place between August 16, 2018 and August 24, 2018. Tables containing more details about the conditions in which the training methods were tested are included in Appendices E and F.

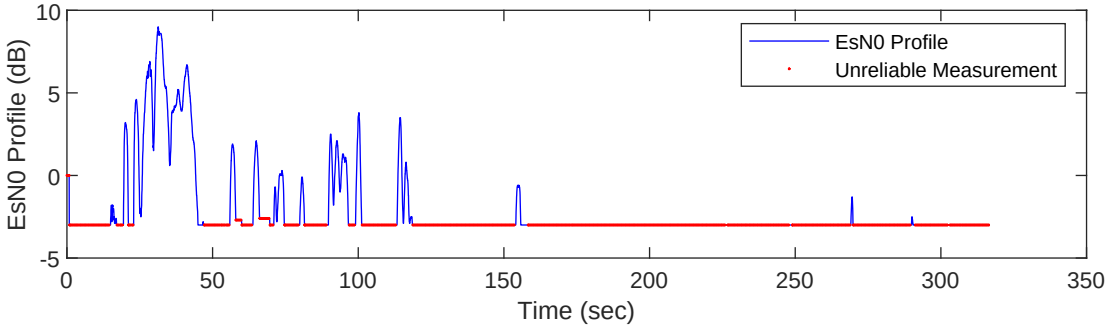
The quality of passes were split into qualitative categories: Excellent/Great, Good, and OK/Poor. Excellent and Great (as well as OK and Poor) are separate categories, but have been combined during processing as a result of the limited amount of time for testing. As stated previously, these categories were determined by the NASA flight staff, and were related to the amount of time where the E_s/N_0 was above certain thresholds. Observed E_s/N_0 profiles representing each category are shown in Figure 4.13.



(a) E_s/N_0 observerd during flight test II-24, classified as Great.



(b) E_s/N_0 observerd during flight test II-23, classified as Good.



(c) E_s/N_0 observerd during flight test II-19, classified as Poor.

Figure 4.13: E_s/N_0 profiles observed with each category label.

Overview plots of CE-LM, CE-RLM, and CE-NSE are shown in Figures 4.14,4.15 and 4.16, respectively. Like before, it is a bit challenging to make significant statements based on these plots, made even more difficult by the fact that the passes are no longer directly comparable. Nonetheless, there are still a few things to be observed. First of all, in the first portion of the pass (roughly from 0-150 s), CE-LM and CE-NSE both are stable in the action chosen, which improves in fitness score as the E_s/N_0 improves. This is unlike CE-RLM, which appears to periodically get dislodged from an action by a variation in the E_s/N_0's trend upwards. These same patterns do not impact CE-LM or CE-NSE. The next significant portion of the pass (250-500 s) maintains the same behavior from CE-RLM, while CE-LM and CE-NSE no longer have a stable action chosen.

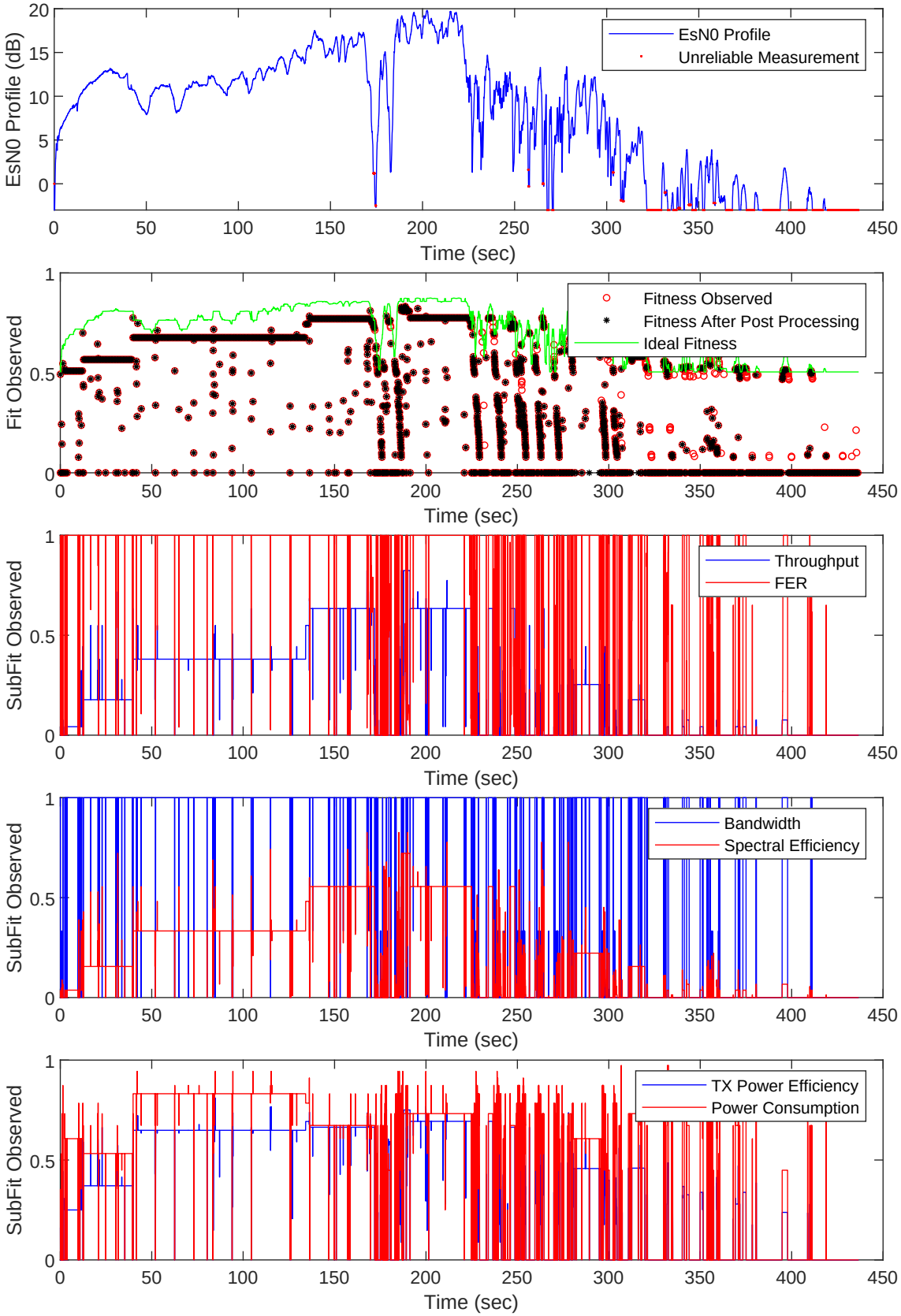


Figure 4.14: Operation of CE-LM during Great quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

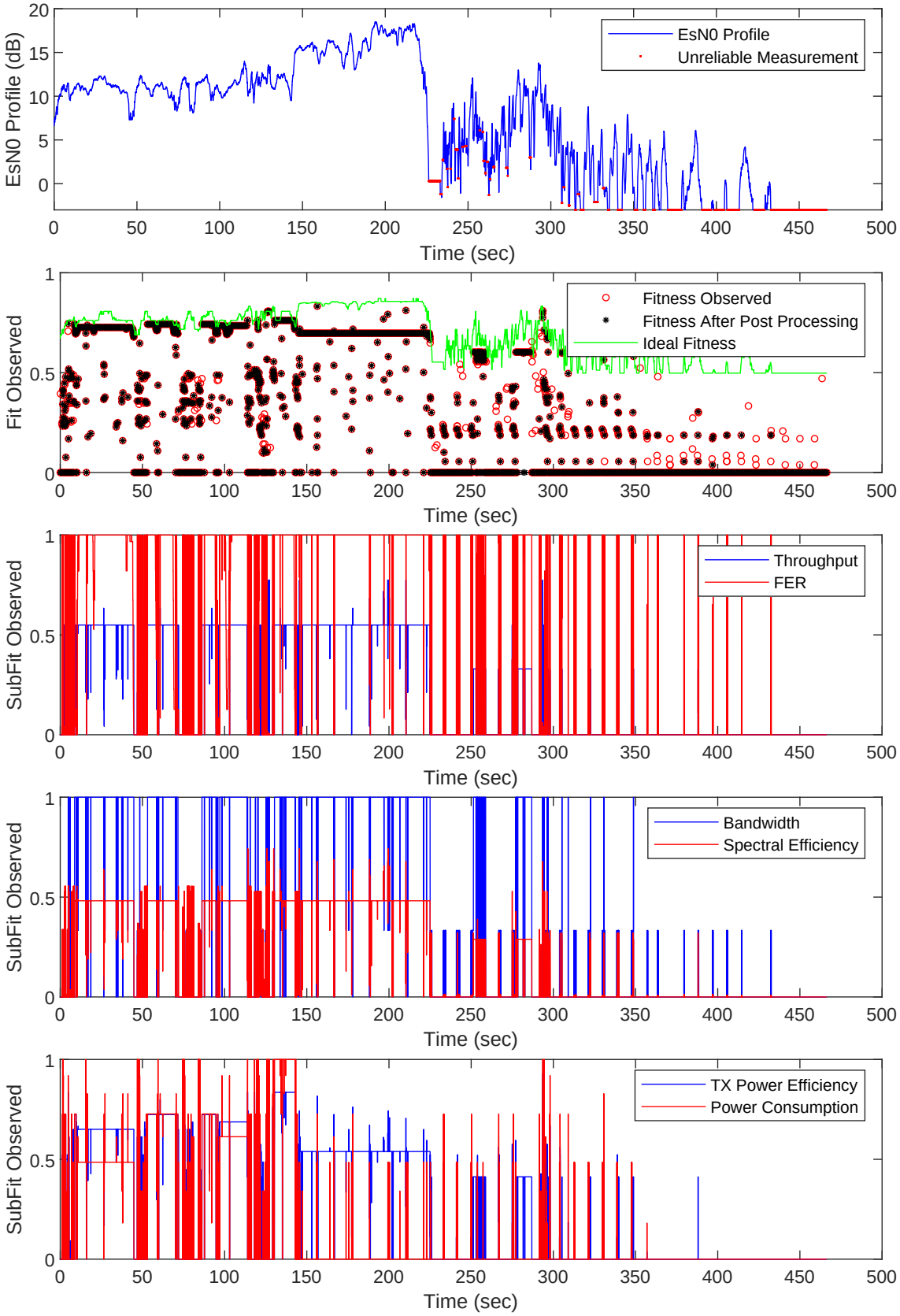


Figure 4.15: Operation of CE-RLM during Great quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

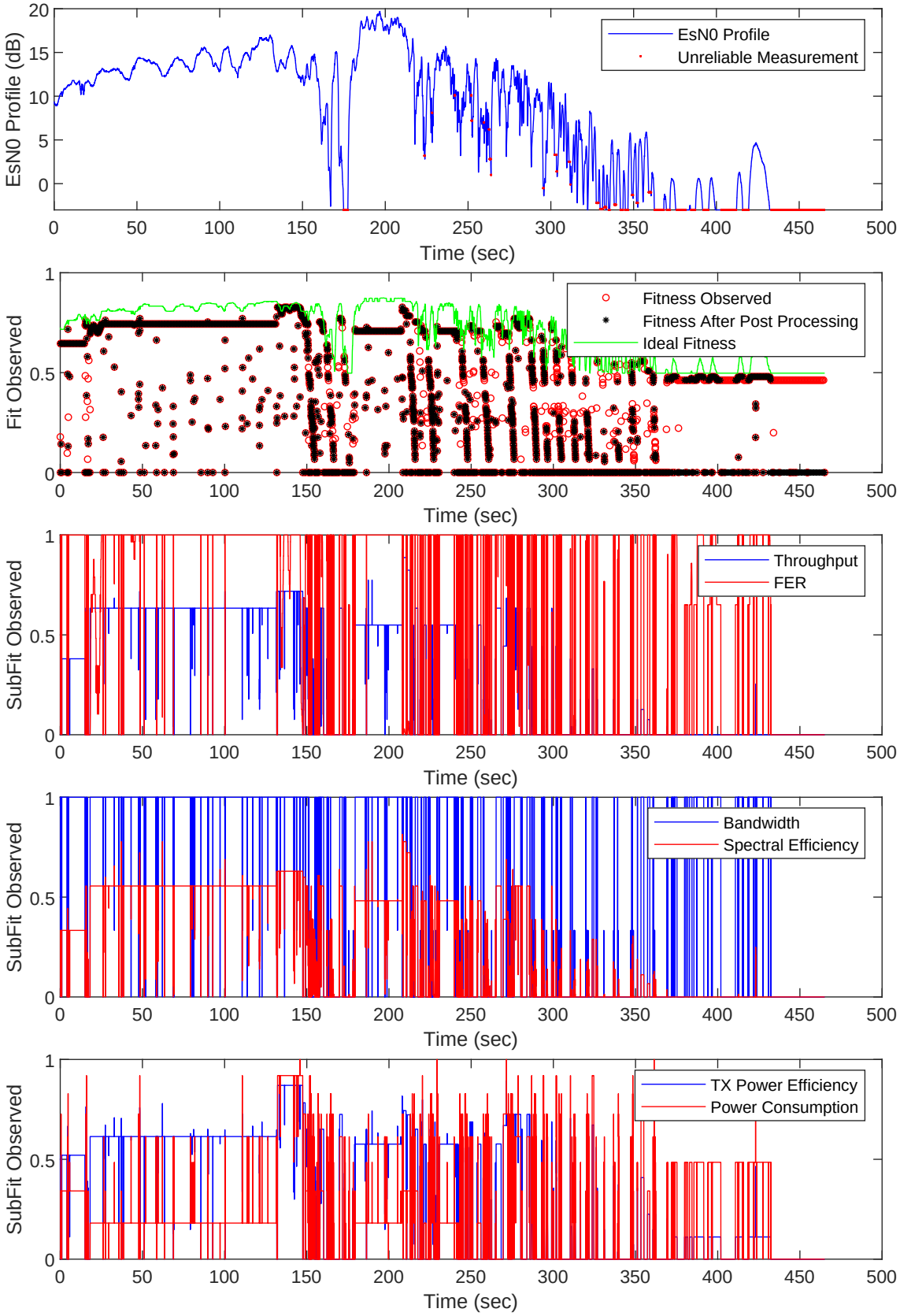
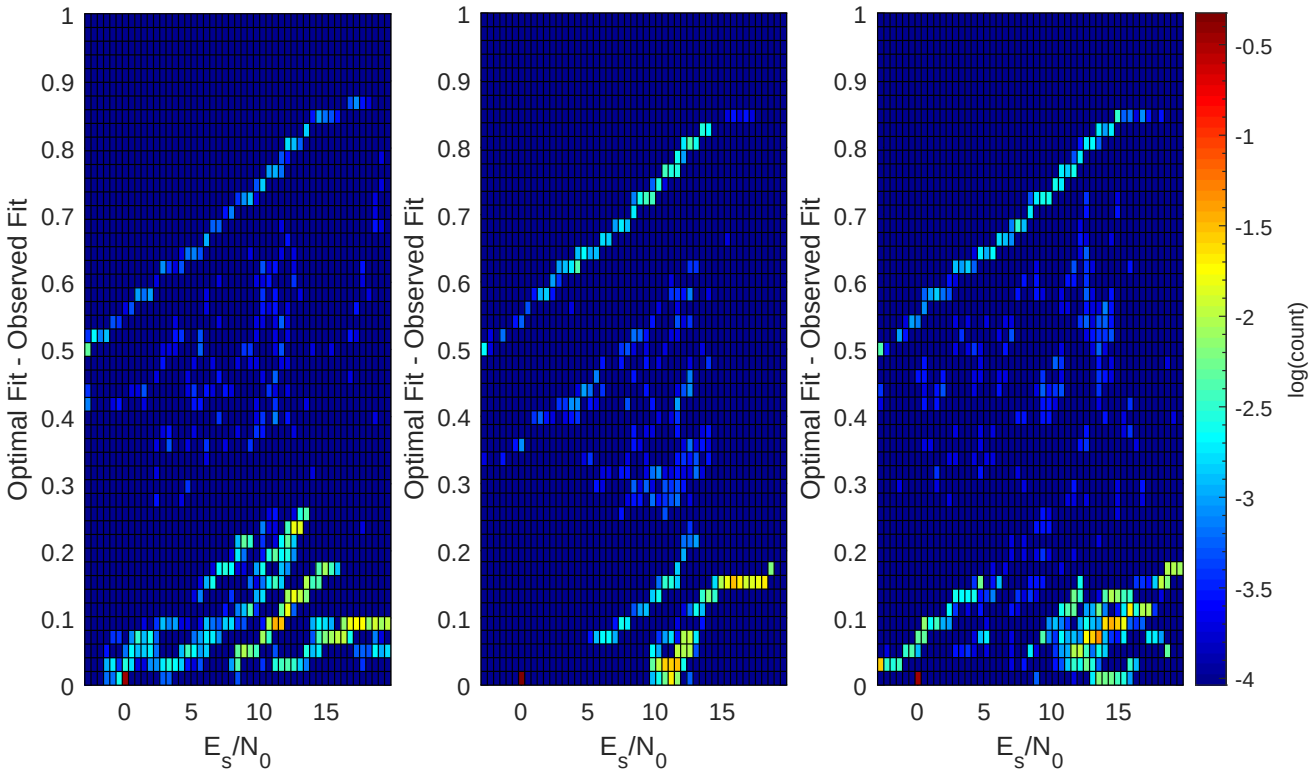


Figure 4.16: Operation of CE-NSE during Great quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

Figure 4.17 shows the 2D histograms generated for each training method. Like the C++ simulation, the X dimension of the histogram is the E_s/N_0 observed, and the Y dimension is the fitness distance observed. However, unlike the C++ simulation, the colored dimension is the count of samples that fall in that bin normalized by the total count of samples. This is done to mitigate the fact that different passes are likely to have different numbers of frames observed before finishing. In the C++ simulation this normalization was less important because the different training methods all used the same E_s/N_0 profile. However, with the flight test results the counts likely to vary by a significant margin



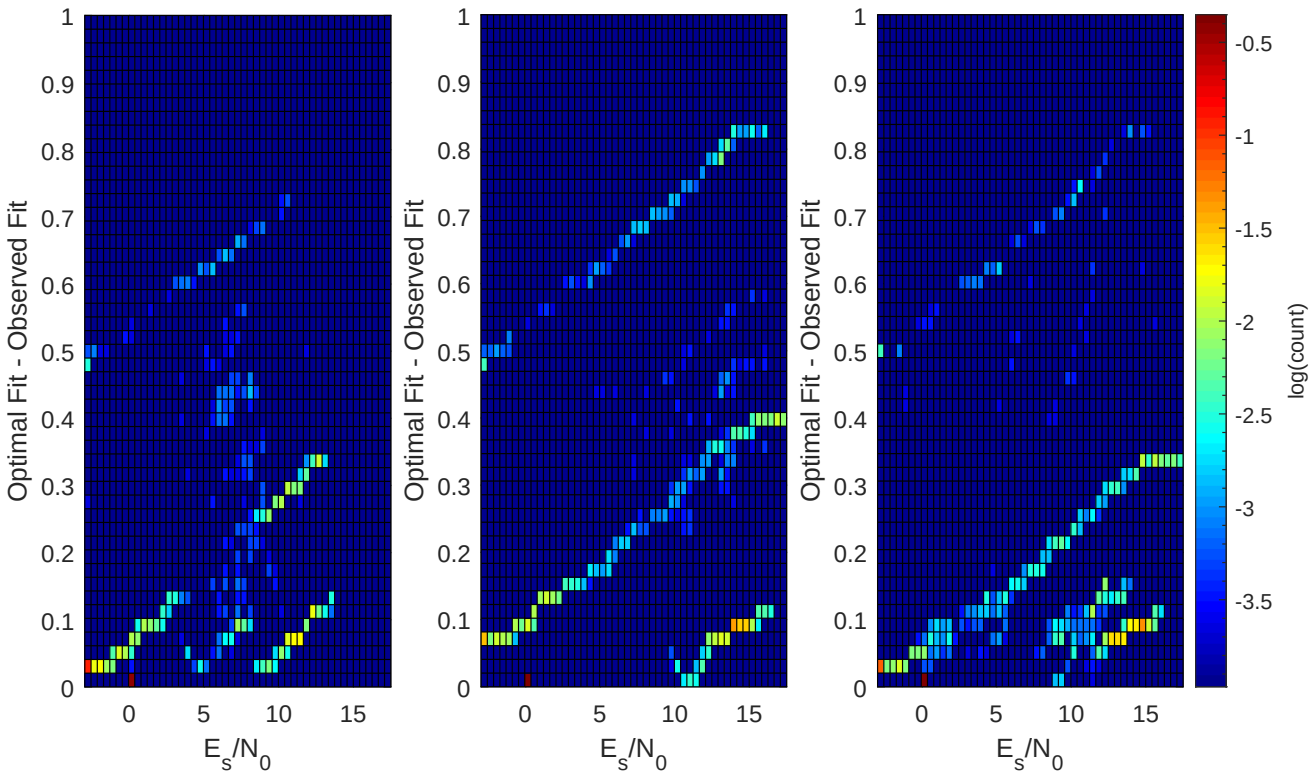
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure 4.17: Two-dimensional histograms of each training method operating the Cooperation mission on a Great quality pass. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed/total number of frames)

Looking at Figure 4.17, the behavior of each training method becomes clear. CE-RLM performs the worst. It appears to have converged fairly well to an action

in the high E_s/N_0 regime. It appears to be locked on to this action more so than CE-LM or CE-NSE, which have a wider range of fitness distances observed in the same regime, all with lower fitness distances. Beyond the middle E_s/N_0 regime though, it loses any action to be taken, and keeps exploring. CE-NSE performs similarly to CE-LM in the higher E_s/N_0 regime and the lower E_s/N_0 regime, but has trouble in the middle regime. Comparing Figures 4.14 and 4.16, it's likely that this is because the E_s/N_0 changes more rapidly during between 250 and 350 seconds. CE-LM appears to lock on to an action during this time, while CE-NSE does not.

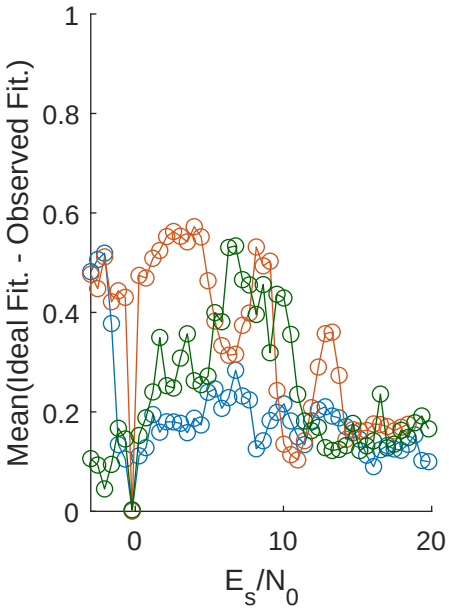
Figure 4.18 shows the 2D histograms for the training methods during Good passes. The histogram for CE-RLM is fairly straightforward, with the high E_s/N_0 regime being from the beginning of the pass, and the middle line representing the action that is chosen during forced retraining. The CE-LM and NSE-LM are similar to the performance during the great pass, except with the roles reversed. This plot is included to provide a point of comparison between performance in Great pass conditions and performance in Good pass conditions. 2D histograms for the rest of the flight conditions are included in Appendix A.2.



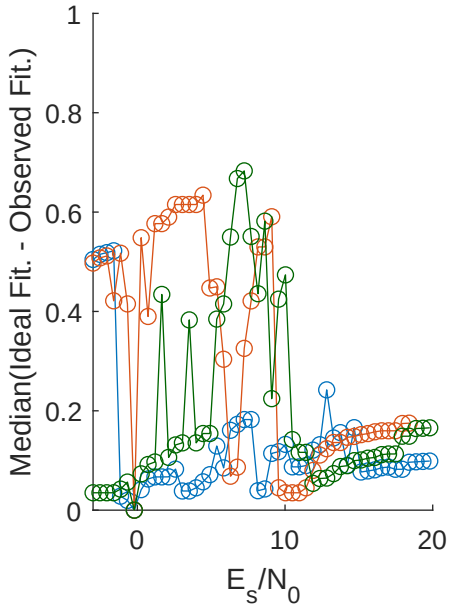
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure 4.18: Two-dimensional histograms of each training method operating the Cooperation mission on a Good quality pass. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed/total number of frames)

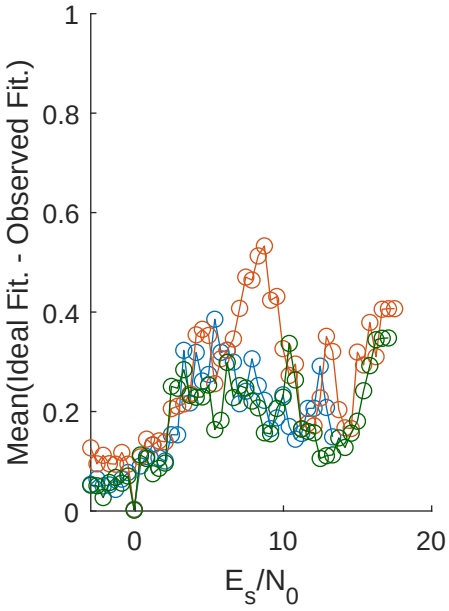
Figure 4.19 shows the binned mean and median values for both the Great passes and the Good passes for the Cooperation mission. The mean values shown in Figures 4.19a show the difference between CE-LM and CE-NSE as primarily in the intermediate E_s/N_0 values. However, by looking at the median values in Figure 4.19b, it's clear that CE-NSE works worse than CE-LM at lower E_s/N_0 values as well. This contrasts with the performance shown in Figure 4.19c, in which the mean values of CE-NSE appear to be slightly better in intermediate E_s/N_0 values than CE-LM. By looking at the median values, this difference is extended.



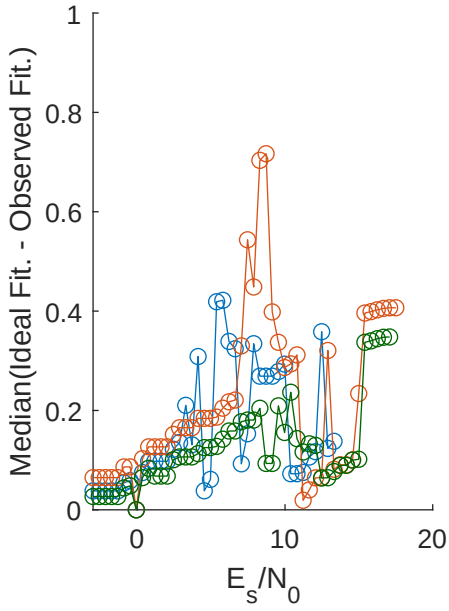
(a) Binned mean for all CE variants during a Great quality pass using Cooperation mission.



(b) Binned median for all CE variants during a Great quality pass using Cooperation mission.



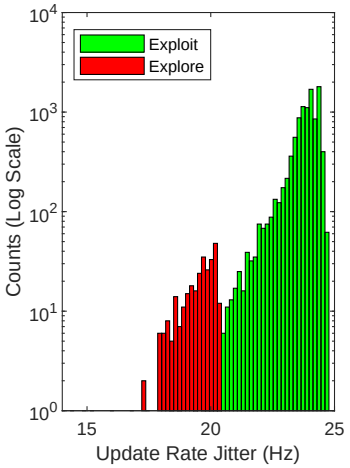
(c) Binned mean for all CE variants during a Good quality pass using Cooperation mission.



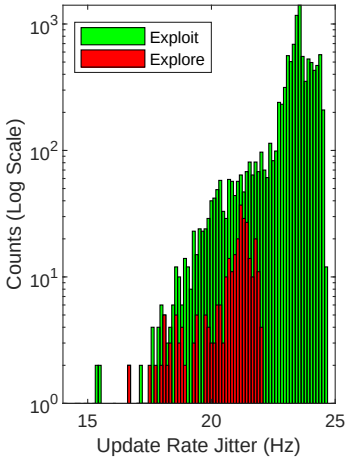
(d) Binned median for all CE variants during a Good quality pass using Cooperation mission.

Figure 4.19: Binned mean and median plots for all CE variants, using Cooperation mission

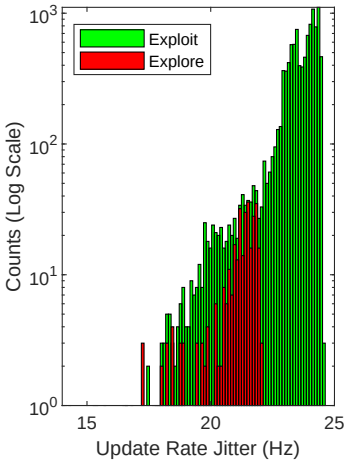
In addition to the performance of the training algorithms, the time the CE takes to choose and transmit an action after receiving data from the satellite is also critical. One-dimesnional histograms of this update rate for both Explore and Exploit networks are shown in Figure 4.20. These histograms imply that there isn't a meaningful difference in execution time. This can be attributed to the fact that CE-LM, CE-RLM, and CE-NSE all have the same underlying MLP architecture, and so the actual application of the MLPs don't take any longer beyond any difference in number of members in the ensemble. At the relatively shallow level of the MLPs used in the CE and the small ensembles, the difference was negligible in execution. Because of this, the major differentiation in execution time is related to how long it takes for each training method to be retrained.



(a) Histogram of time it takes to predict with MLPs, CE-LM.



(b) Histogram of time it takes to predict with MLPs, CE-RLM.

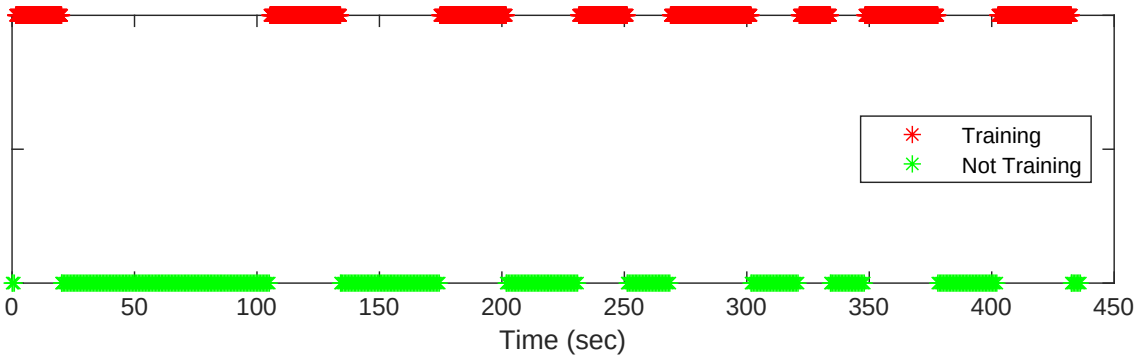


(c) Histogram of time it takes to predict with MLPs, CE-NSE.

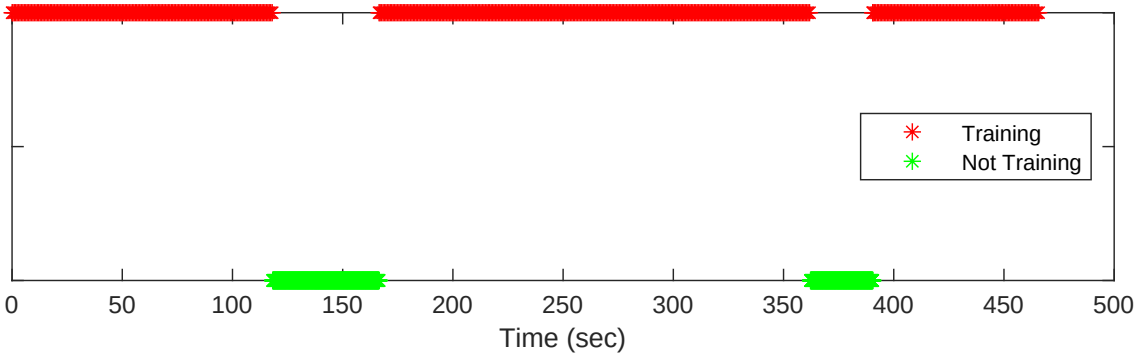
Figure 4.20: Histograms illustrating time it takes to predict for each CE variant.

A time series outlining how long the CE was in training mode for each run is shown in Figure 4.21. It is in this plot where the big time issue with CE-RLM becomes evident. The fact that RLM takes a highly parallel operation and makes it sequential is evident by the fact that it takes 200 seconds to train on one buffer. Given the fact that a single pass tends to take at most around 500 seconds, this training time is too long, and prevents the network from adapting at a useful rate. This is what can be attributed for the passes in which CE-RLM fails to converge appropriately.

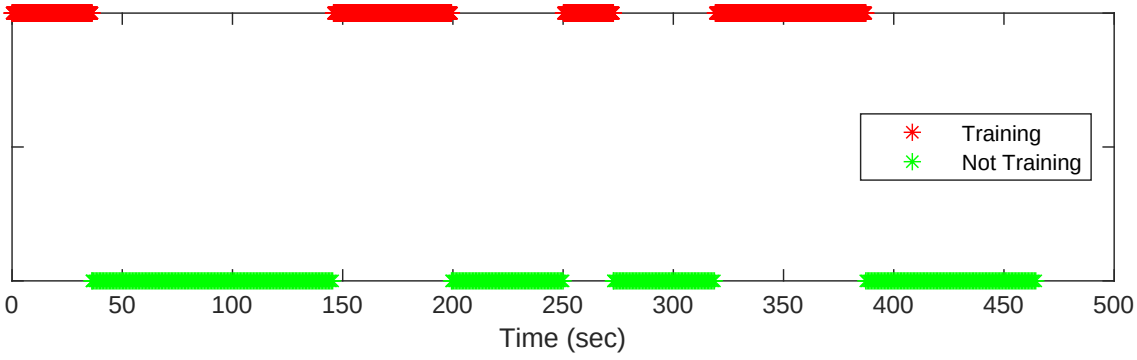
The comparison between training time for CE-LM and CE-NSE is more suitable than the comparison between CE-LM and CE-RLM, because CE-NSE uses LM when training the individual networks of the ensemble. As such, it is unsurprising that CE-LM takes less time to train than CE-NSE, which has additional calculations in addition to the LM training. The small number of results from the flight tests make any further analysis of timing beyond this level unlikely to provide useful insight. The ultimate conclusion is that CE-NSE, while taking more time to train than CE-LM, still remains within the realm of usability. The same cannot be said for CE-RLM.



(a) Plot illustrating time CE-LM spent training vs not training.



(b) Plot illustrating time CE-RLM spent training vs not training.



(c) Plot illustrating time CE-NSE spent training vs not training.

Figure 4.21: Time series illustrating how much time each CE variant uses training vs not training during a Great quality pass while using the Cooperation mission.

4.4 Summary

In this section, the results from the tests conducted for evaluating training methods that mitigate Catastrophic Forgetting are explored. First, the MATLAB simulation used to verify the validity of CE-RLM and CE-NSE. Following this, the simulation results from the C++ implementation of the CE are explored, as they allow for the different training methods be run on the same E_s/N_0 profile. Finally, the results from the flight test were analyzed. Overall, CE-NSE appeared to be the better solution to Catastrophic Forgetting than CE-RLM, which takes much more time to train and converge to a behavior pattern. On casual inspection it is difficult to determine if CE-NSE outperforms CE-LM. However, by looking at the binned mean and median values over the simulations, CE-NSE repeatedly chooses actions closer to the ideal fitness score than CE-LM does.

Chapter 5

Conclusion and Future Work

In this thesis, the cognitive engine developed for space communications was modified to address the problem of Catastrophic Forgetting. After testing and implementation, the modifications proved to improve performance of the CE. In the following section, the salient achievements of the thesis are described and potential future work is provided.

5.1 Research Achievements

Based on the current state-of-the-art and the work conducted by [2] and [1], there were two main extensions provided by this thesis:

- **CE-NSE mitigated some effects of Catastrophic Forgetting, and outperformed the baseline CE:** Cognitive engines for autonomous space communications that address Catastrophic Forgetting were developed and tested onboard the ISS. The performance of both CE-RLM and CE-NSE were compared to the baseline CE-LM implementation tested in [1]. CE-NSE proved to provide modest improvements in actions chosen without imposing an unreasonble training time penalty.

- **GANs were determined to not be directly applicable to the CE architecture:** The concept of two networks collaboratively working together was present in both the CE and GANs. However, the low supply of offline training data for the GAN makes it unlikely that any GAN employed would be trained sufficiently to have converged on the proper data distribution. In addition, the GAN would likely have to be supplementary to the RLNN structure, instead of replacing the Explore and Exploit networks.

5.2 Future Work

- One of the goals that did not have time to be completed was the extension of the CE to MAC layer parameters. Adding this layer greatly increases the action space possible and the evaluation metrics that must be balanced. Integrating this aspect was deemed too complex for this thesis, but could provide significant imporvements.

- The implementation of CE-NSE simply prunes the ensemble of MLPs by getting rid of the oldest one, under the assumption that the oldest MLP will be the least relevant to the current set of data. This assumption is not necessarily correct. Using a smarter pruning method could continue to improve the results of the CE.

- CE-NSE uses MSE when evaluating the performance of the individual MLPs in the ensemble. This may not be the best way to evaluate the performance of the MLPs. Instead, a metric more relevant to Satellite Communications may be more relevant.

- Pretraining was conducted using a small subset of SNR profiles, in the simplest manner possible. A more priniciple building of an ensemble (such as training

individual networks on different portions of an SNR profile that are common) could provide additional improvements.

- Details about the MLPs being used in the CE (like number of hidden layers and number of nodes in the layers) were determined using a simple SNR profile that is not very representatitve of the channel conditions for satellite communications. Re-evaluating these chosen parameters would likely improve the performance of the CE in the more difficult situations it encounters.

Chapter 6

Bibliography

[1] T. M. Hackett, S. G. Bilén, P. V. R. Ferreira, A. M. Wyglinski, and R. C. Reinhart. Implementation of a space communications cognitive engine. In *2017 Cognitive Communications for Aerospace Applications Workshop (CCAA)*, pages 1–7, June 2017.

[2] P. V. R. Ferreira et al. Multi-objective reinforcement learning-based deep neural networks for cognitive space communications. In *2017 Cognitive Communications for Aerospace Applications Workshop (CCAA)*, pages 1–8, June 2017.

[3] Tim Hwang. Computational Power and the Social Impact of Artificial Intelligence. *ArXiv e-prints*, page arXiv:1803.08971, Mar 2018.

[4] Maximillian C. Scardelletti et al. Software defined radio architecture for nasa's space communications. *High Frequency Electronics*, 6(7):18–25, 2007-07.

[5] C. Morlet. Software radio in space segment: applications, technologies, reconfiguration management and architectures. pages 269,270. IEEE, 2006-09.

[6] *Software defined radio architectures, systems, and functions*. Wiley series in software radio. Wiley, Hoboken, NJ, 2003.

[7] Andrew Silver. Luxembourg invests (euro)25 million in asteroid mining. `https://spectrum.ieee.org/tech-talk/aerospace/space-flight/luxembourg-invests-25-million-in-asteroid-mining`, 2016. Accessed: 2018-11-15.

[8] Ferrucci, David et al. Building watson: An overview of the deepqa project. *AI Magazine*, 31(3):59,79, 2010-10-01.

[9] David Silver et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016-01-27.

[10] Georgios Gardikis, Nikolaos Zotos, and Anastasios Kourtis. Satellite media broadcasting with adaptive coding and modulation. *International Journal of Digital Multimedia Broadcasting*, 2009(2009):1–10, 2009-01-01.

[11] A.J. Goldsmith and S.-G. Chua. Adaptive coded modulation for fading channels. *Communications, IEEE Transactions on*, 46(5):595,602, 1998-05.

[12] Daniele Tarchi, Giovanni E. Corazza, and Alessandro Vanelli-Coralli. Adaptive coding and modulation techniques for next generation hand-held mobile satellite communications. pages 4504,4508. IEEE, 2013-06.

[13] M. Emmelmann and H. Bischl. An adaptive mac layer protocol for atm-based leo satellite networks. volume 4, pages 2698,2702, USA, 2003. IEEE.

[14] European Telecommunications Standards Institute. Etsi en 302 307-1: Digital video broadcasting (dvb); second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broadband satellite applications; part 1: Dvb-s2. Standard, European Telecommunications Standards Institute, 2014.

[15] M.M. Alani. Osi model. In *SpringerBriefs in Computer Science*, number 9783319051512, pages 5,17. Springer, 2014.

[16] Aldo Cugnini. Mpeg-4 avc. *Broadcast Engineering*, 52(8):n/a, 2010-08-01.

[17] European Telecommunications Standards Institute. Etsi en 302 307-2: Digital video broadcasting (dvb); second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broadband satellite applications; part 2: Dvb-s2 extensions (dvb-s2x). Standard, European Telecommunications Standards Institute, 2014.

[18] James F. Kurose. *Computer networking : a top-down approach*. Addison-Wesley, Boston, 5th ed. edition, 2010.

[19] Ekram Hossain, Dusit Niyato, and Dong In Kim. Evolution and future trends of research in cognitive radio: a contemporary survey. *Wireless Communications and Mobile Computing*, 15(11):1530,1564, 2015-08-10.

[20] Si Chen, Timothy R. Newman, Joseph B. Evans, and Alexander M. Wyglinski. Genetic algorithm-based optimization for cognitive radio networks. pages 1,6. IEEE Publishing, 2010-04.

[21] He, An et al. A survey of artificial intelligence for cognitive radios. *Vehicular Technology, IEEE Transactions on*, 59(4):1578,1592, 2010-05.

[22] Mario Bkassiny, Sudharman K. Yang Li, and Sudharman K. Jayaweera. A survey on machine-learning techniques in cognitive radios. *Communications Surveys & Tutorials, IEEE*, 15(3):1136,1159, 2013.

[23] Khashayar Kotobi and Sven G. Bilen. Introduction of vigilante players in cognitive networks with moving greedy players. pages 1,2. IEEE, 2015-09.

[24] Symeon. Chatzinotas. *Cooperative and cognitive satellite systems*. Academic Press, Amsterdam, [Netherlands], 2015.

[25] Khashayar Kotobi, Philip B. Mainwaring, and Sven G. Bilen. Puzzle-based auction mechanism for spectrum sharing in cognitive radio networks. pages 1,6. IEEE, 2016-10.

[26] Boutaba, Raouf et al. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):16, Jun 2018.

[27] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen A.W.M. van Der Laak, Bram van Ginneken, and Clara I. SáNchez. A survey on deep learning in medical image analysis. *Medical Image Analysis*, 42(C):60,88, 2017-12.

[28] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234(C):11,26, 2017-04-19.

[29] Nils J. Nilsson. Introduction to machine learning. Web-published, 2005.

[30] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, second edition, 2017.

[31] W. G. Hatcher and W. Yu. A survey of deep learning: Platforms, applications and emerging research trends. *IEEE Access*, 6:24411–24432, 2018.

[32] Michael A Nielsen. Sigmoid neurons. In *Neural Networks and Deep Learning* [74], chapter 1.

[33] Rohan Varma. Picking loss functions - a comparison between mse, cross entropy, and hinge loss, 2018.

[34] Michael A Nielsen. How the backpropagation algorithm works. In *Neural Networks and Deep Learning* [74], chapter 2.

[35] Ake Bjorck. *Numerical methods for least squares problems*, volume 51. Siam, 1996.

[36] H. Yu and B.M. Wilamowski. Levenberg-marquardt training. In *Intelligent Systems*. CRC Press, 2016-04-19.

[37] B. M. Wilamowski and H. Yu. Improved computation for levenberg–marquardt training. *IEEE Transactions on Neural Networks*, 21(6):930–937, June 2010.

[38] Sebastian Ruder. An overview of gradient descent optimization algorithms. 2016-09-15.

[39] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014-12-22.

[40] Andrej. Karpathy. Learning and evaluation. `http://cs231n.github.io/neural-networks-3/`, 2018. Accessed: 2018-12-14.

[41] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[42] Michel Tokic. Adaptive ε-greedy exploration in reinforcement learning based on value differences. In *Annual Conference on Artificial Intelligence*, pages 203–210. Springer, 2010.

[43] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *ML*, 1990.

[44] Richard Reinhart and James P Lux. Space-based reconfigurable software defined radio test bed aboard international space station. In *SpaceOps 2014 conference*, page 1612, 2014.

[45] P Bricker, J Luecke, and D Herr. Integrated receiver for nasa tracking and data relay satellite system. In *Military Communications Conference, 1990. MILCOM'90, Conference Record, A New Era. 1990 IEEE*, pages 1–5. IEEE, 1990.

[46] Richard C Reinhart, David J Israel, James P Lux, and Andrew L. Benjamin. Space telecommunications radio system strs architecture standard. Technical report, March 2010.

[47] Richard C. Reinhart and Bryan K. Smith. Using international space station for cognitive system research and technology with space-based reconfigurable software defined radios. IAF, 2016-10.

[48] Paulo Ferreira. *SRML: Space Radio Machine Learning*. PhD thesis, Worcester Polytechnic Institute, 2017.

[49] NASA. Ta 5: Communications, navigation, and orbital debris tracking and characterization systems. Technical report, July 2015.

[50] G. Ditzler and R. Polikar. Semi-supervised learning in nonstationary environments. In *The 2011 International Joint Conference on Neural Networks*, pages 2741–2748, July 2011.

[51] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[52] Ian J. Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. 2013-12-21.

[53] Lester SH Ngia, Jonas Sjoberg, and Mats Viberg. Adaptive neural nets filter using a recursive levenberg-marquardt search direction. In *Signals, Systems & Computers, 1998. Conference Record of the Thirty-Second Asilomar Conference on*, volume 1, pages 697–701. IEEE, 1998.

[54] G. Ditzler, M. Roveri, C. Alippi, and R. Polikar. Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine*, 10(4):12–25, Nov 2015.

[55] Dimitri P Solomatine and Durga L Shrestha. Adaboost. rt: a boosting algorithm for regression problems. *Neural Networks*, 2:1163–1168, 2004.

[56] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[57] Ryan Elwell and Robi Polikar. Incremental learning in nonstationary environments with controlled forgetting. pages 771,778. IEEE Publishing, 2009-06.

[58] Dragos D Margineantu and Thomas G Dietterich. Pruning adaptive boosting. In *ICML*, volume 97, pages 211–218, 1997.

[59] Antonia Creswell et al. Generative adversarial networks: An overview. *CoRR*, abs/1710.07035, 2017.

[60] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[61] J Sola and Joaquin Sevilla. Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science*, 44(3):1464–1468, 1997.

[62] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[63] Salimans, Tim et al. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pages 2234–2242, 2016.

[64] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv.org*, 2017-12-06.

[65] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[66] V. Dumoulin, I. Belghazi, B. Poole, O. Mastropietro, A. Lamb, M. Arjovsky, and A. Courville. Adversarially Learned Inference. *ArXiv e-prints*, June 2016.

[67] D. Nguyen and B. Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. pages 21,26. IEEE Publishing, 1990.

[68] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *The Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, March 2010.

[69] Implementation of incremental learning algorithms. `https://github.com/gditzler/IncrementalLearning`. Accessed: 2018-12-10.

[70] Curtin, Ryan R. et al. Mlpack: A scalable c++ machine learning library. *J. Mach. Learn. Res.*, 14(1):801–805, March 2013.

[71] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, 2016.

[72] Boost c++ libraries. `https://www.boost.org/`, 2017.

[73] *MPEG-4 : proceedings of Workshop and Exhibition on MPEG-4 : June 18-20, 2001, San Jose, California, USA*. IEEE, 2002.

[74] Michael A Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[75] T. Hill and P. Lewicki. *Popular Decision Tree: Classification and Regression Trees (C&RT)*, chapter 9. Statsoft, Inc., 2013.

[76] L.S.H. Ngia, J. Sjoberg, and M. Viberg. Adaptive neural nets filter using a recursive levenberg-marquardt search direction. volume 1, pages 697,701. IEEE Publishing, 1998.

[77] V.S. Asirvadam, S.F. Mcloone, and G.W. Irwin. Parallel and separable recursive levenberg-marquardt training algorithm. volume 2002-, pages 129,138, USA, 2002. IEEE.

[78] A. Gepperth and B. Hammer. Incremental learning algorithms and applications. In *2016 European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pages 357–368, April 2016.

[79] Multiple classifier systems : first international workshop, mcs 2000, cagliari, italy, june 21-23, 2000 : proceedings. In *Multiple classifier systems : first international workshop, MCS 2000, Cagliari, Italy, June 21-23, 2000 : proceedings*, Lecture notes in computer science, 1857, Berlin ;, 2000. Springer.

[80] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A. Bharath. Generative adversarial networks: An overview. *Signal Processing Magazine, IEEE*, 35(1):53,65, 2018-01.

[81] R. Polikar, J. Byorick, S. Krause, A. Marino, and M. Moreton. Learn++: a classifier independent incremental learning algorithm for supervised neural networks. In *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*, volume 2, pages 1742–1747, 2002.

Appendices

Appendix A

Flight Test Profiles

A.1 Flight Test Time Series

A.1.1 Cooperation Mission

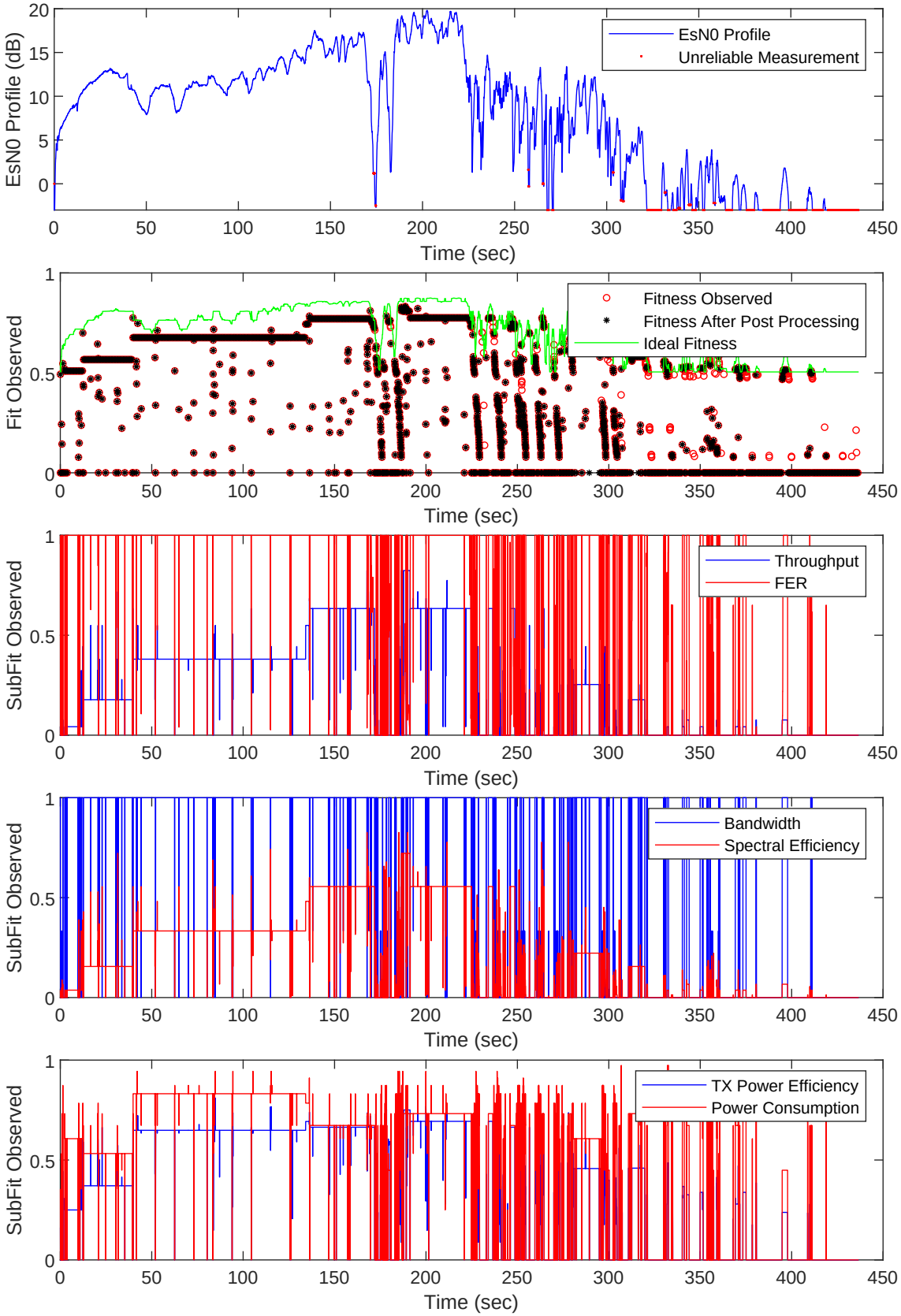


Figure A.1: Operation of CE-LM during Great quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

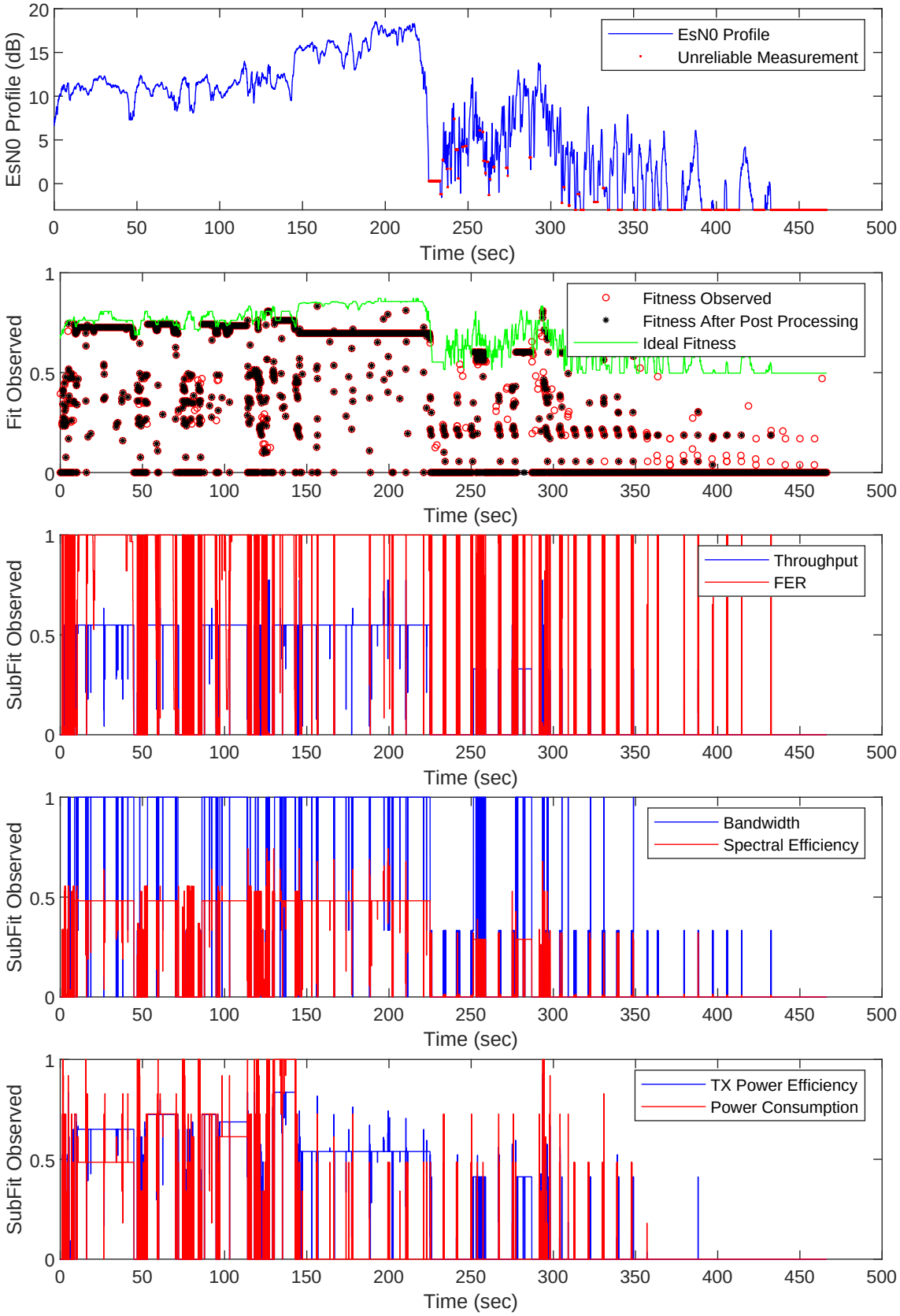


Figure A.2: Operation of CE-RLM during Great quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

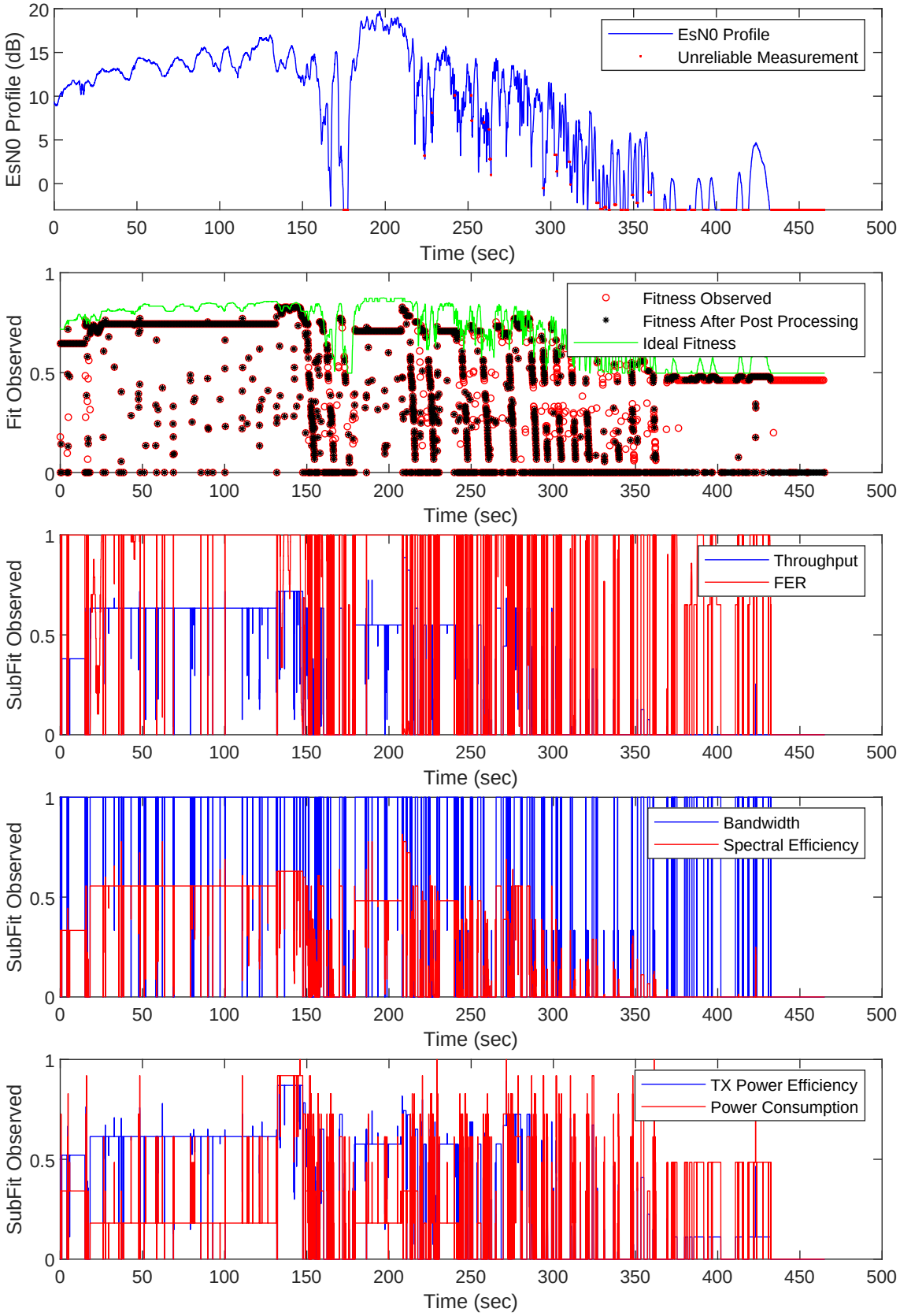


Figure A.3: Operation of CE-NSE during Great quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

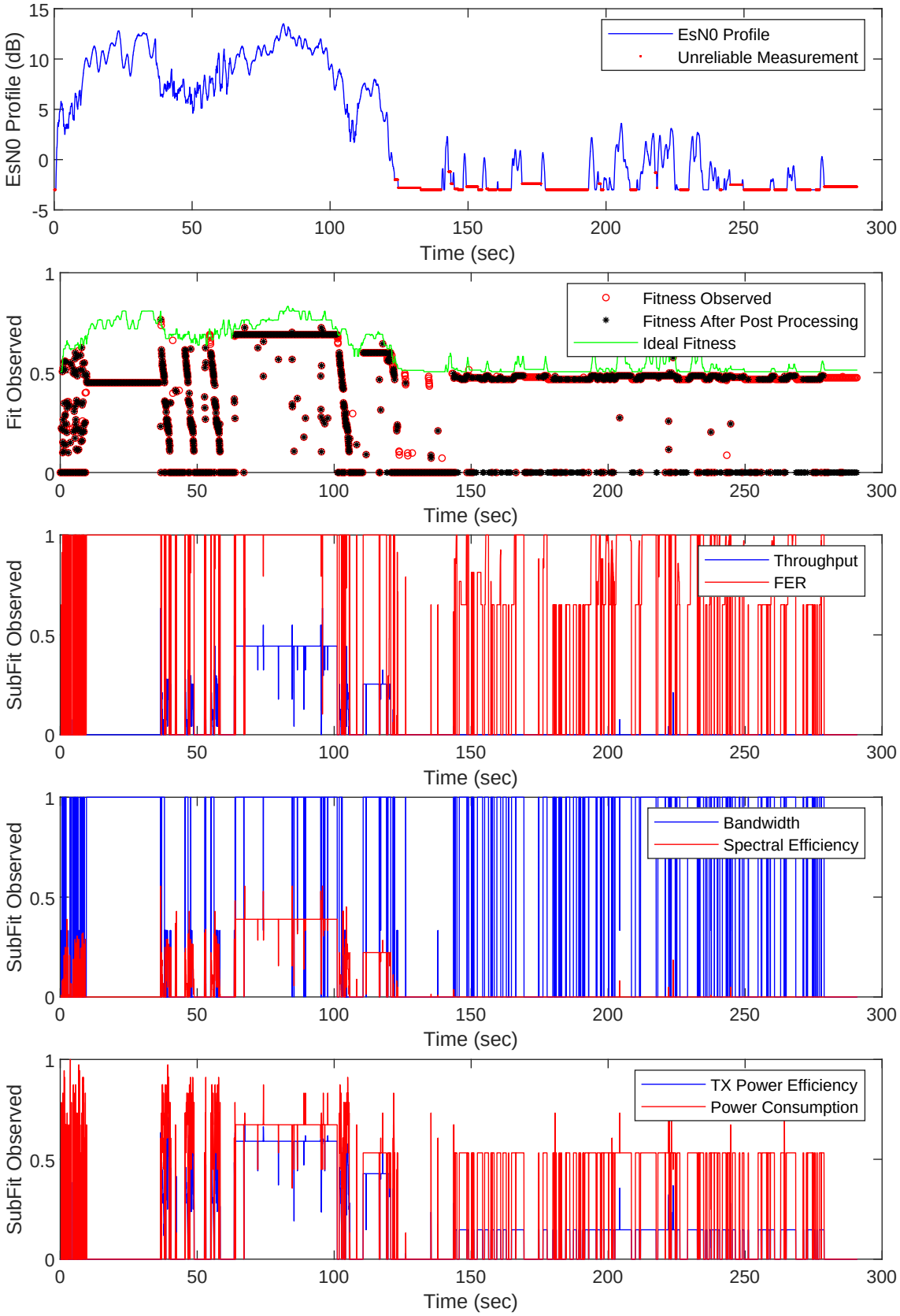


Figure A.4: Operation of CE-LM during Good quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

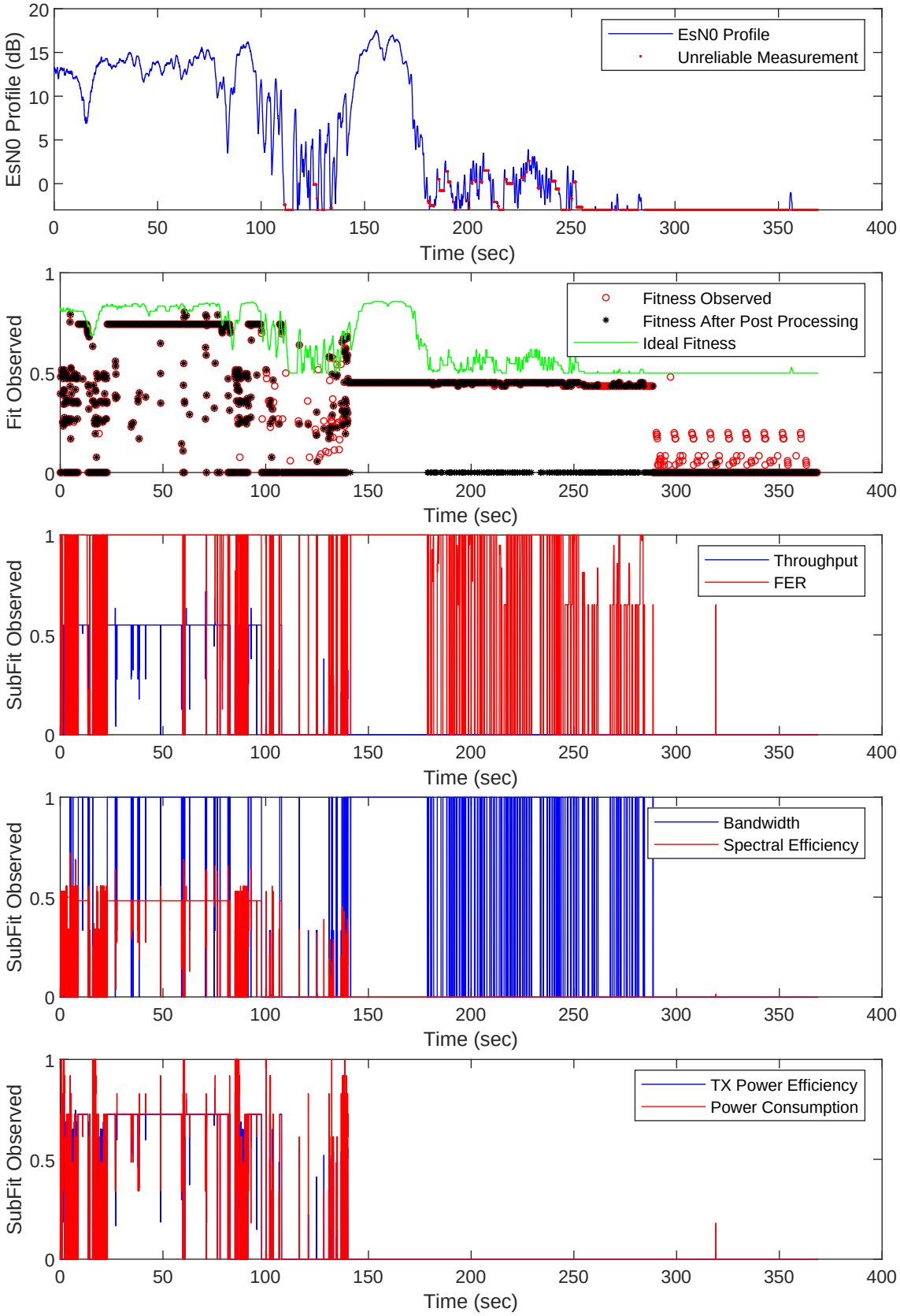


Figure A.5: Operation of CE-RLM during Good quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

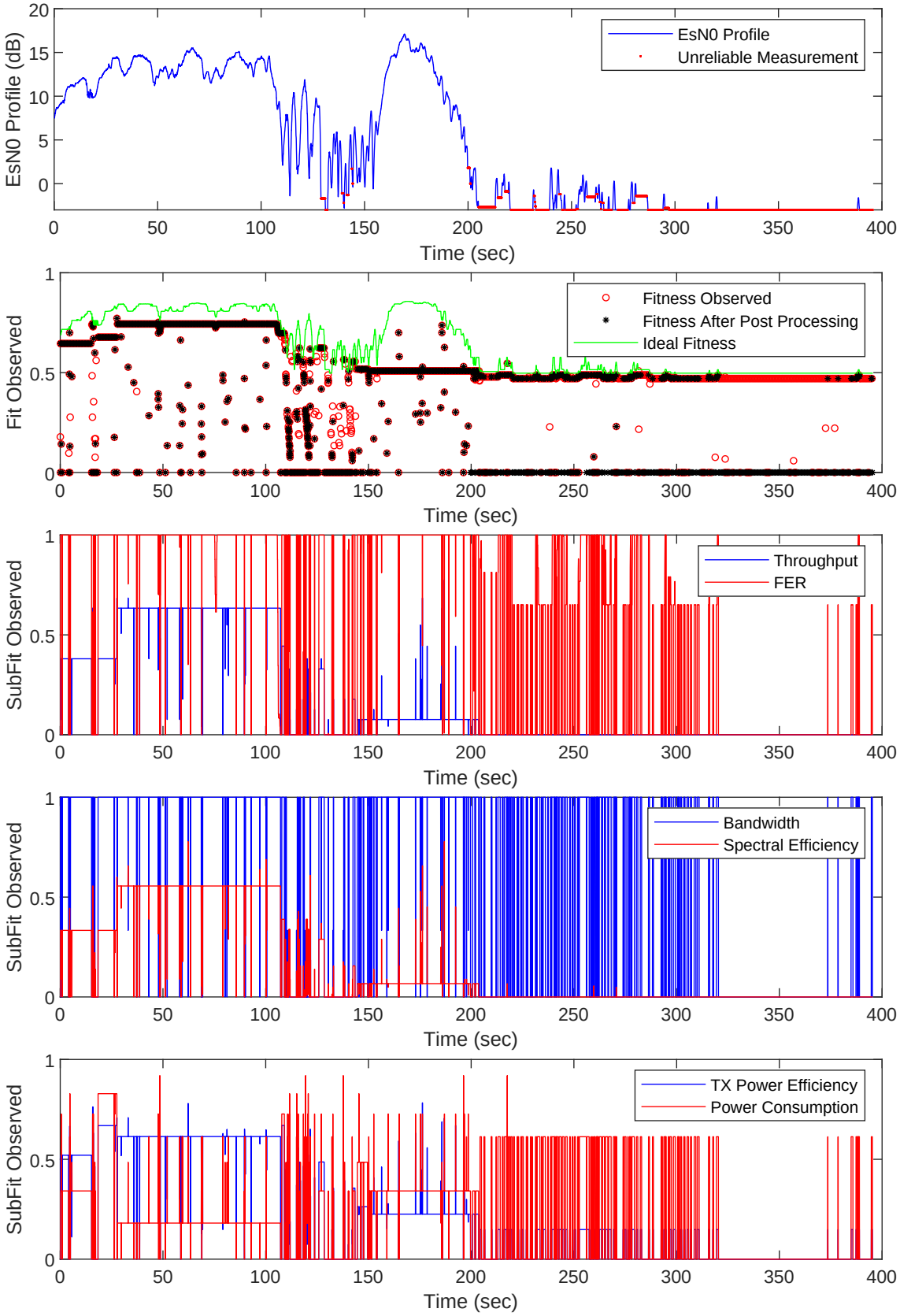


Figure A.6: Operation of CE-NSE during Good quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

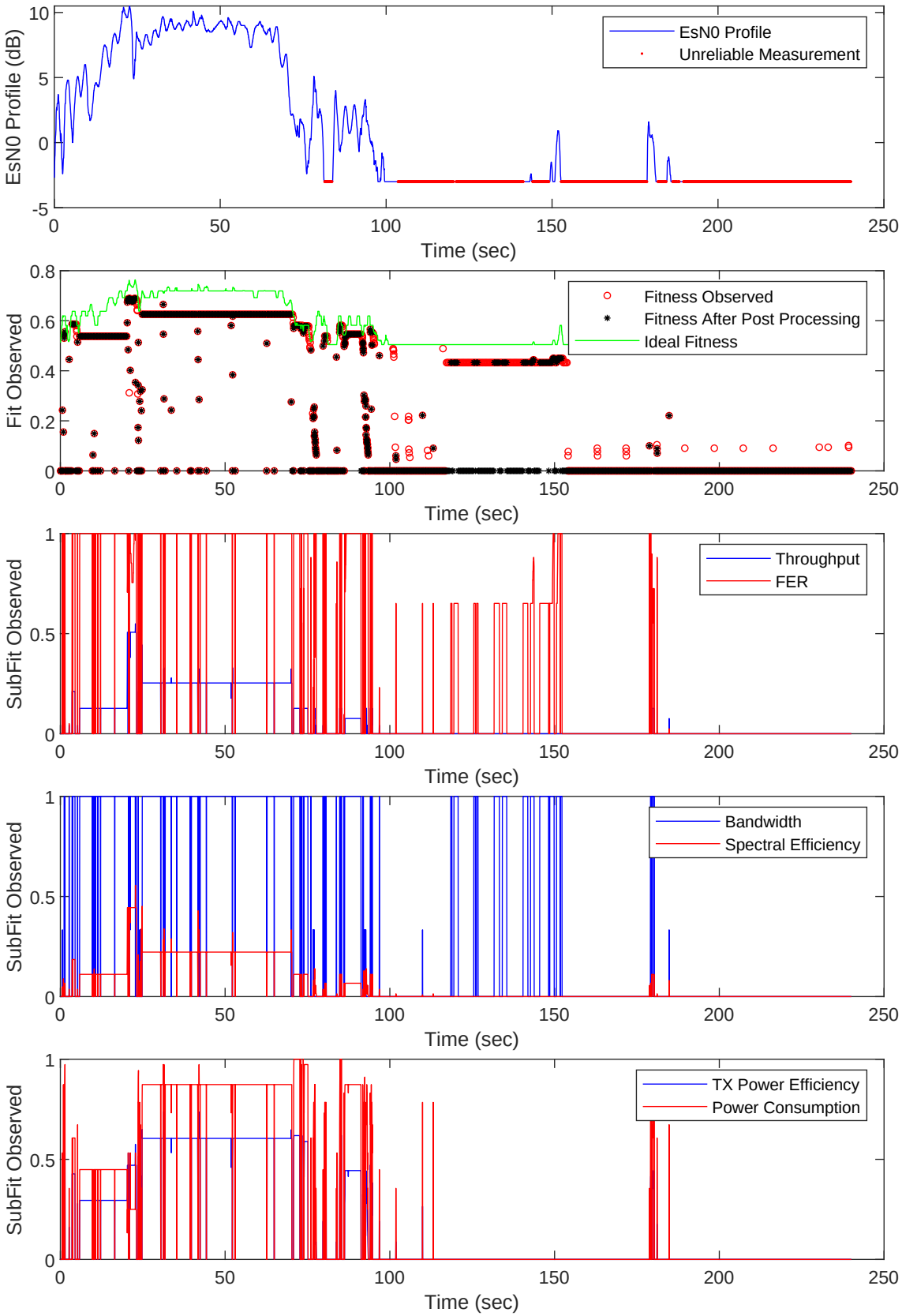


Figure A.7: Operation of CE-LM during Poor quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

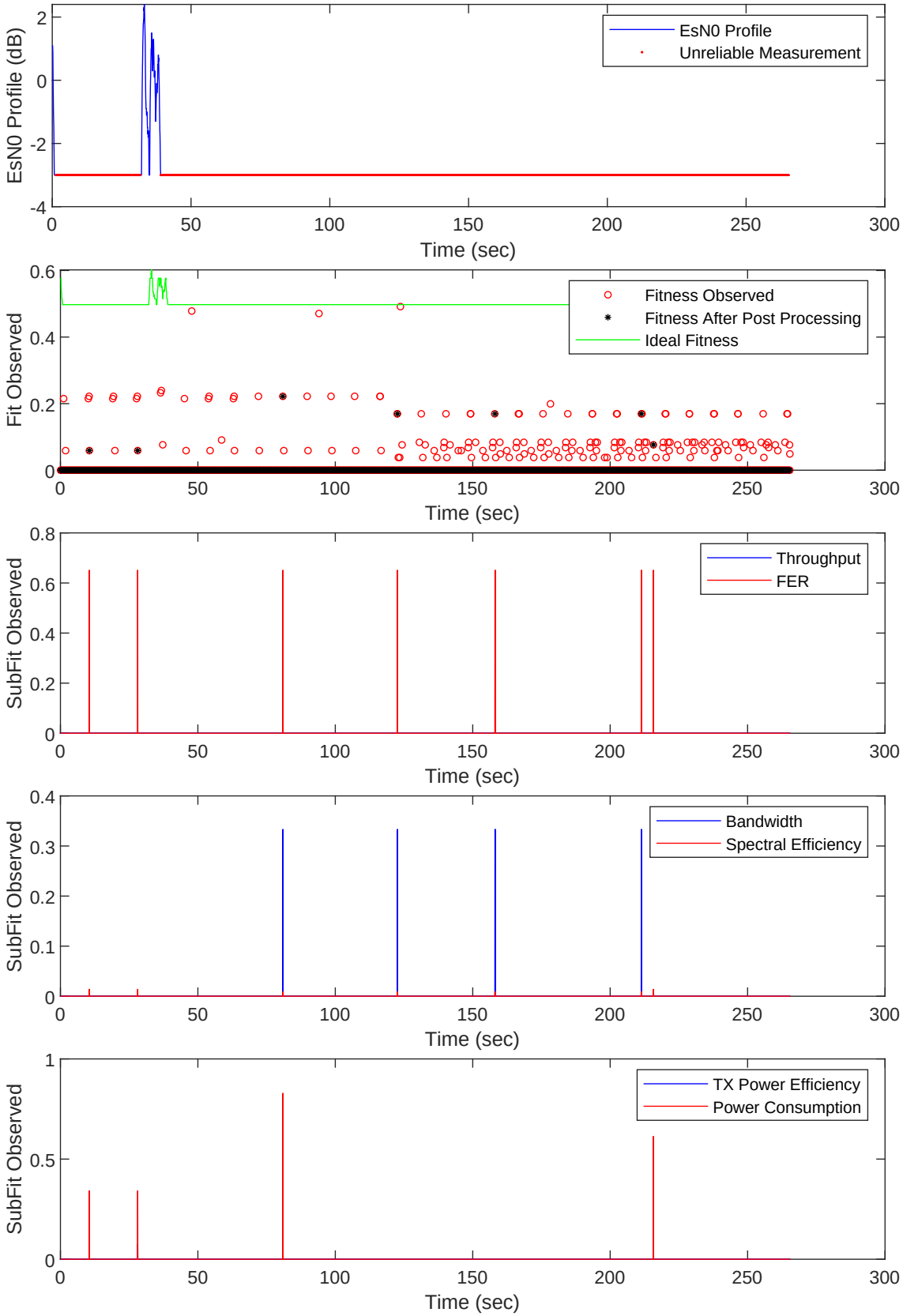


Figure A.8: Operation of CE-RLM during Poor quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

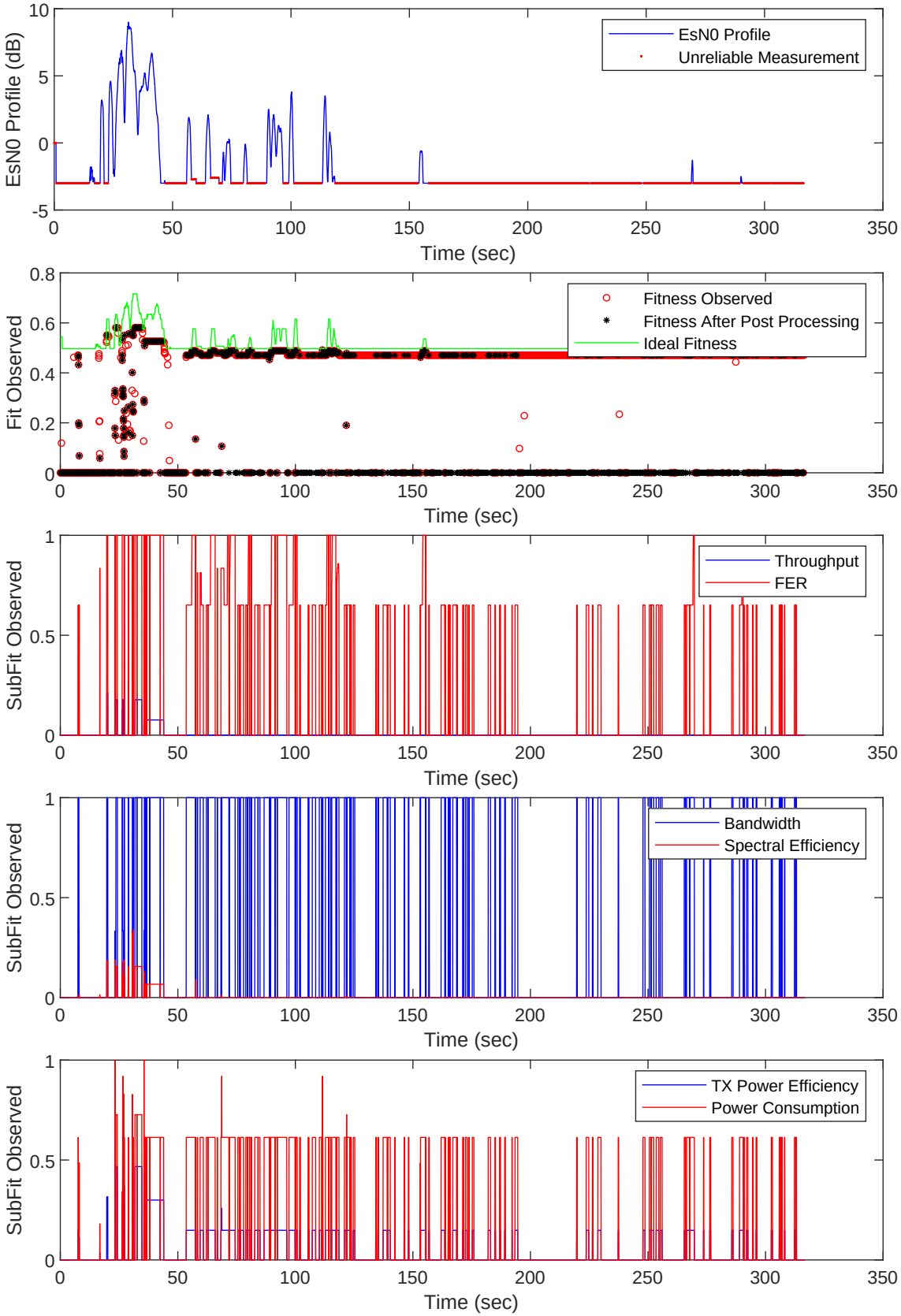


Figure A.9: Operation of CE-NSE during Poor quality pass using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

A.1.2 Powersaving Mission

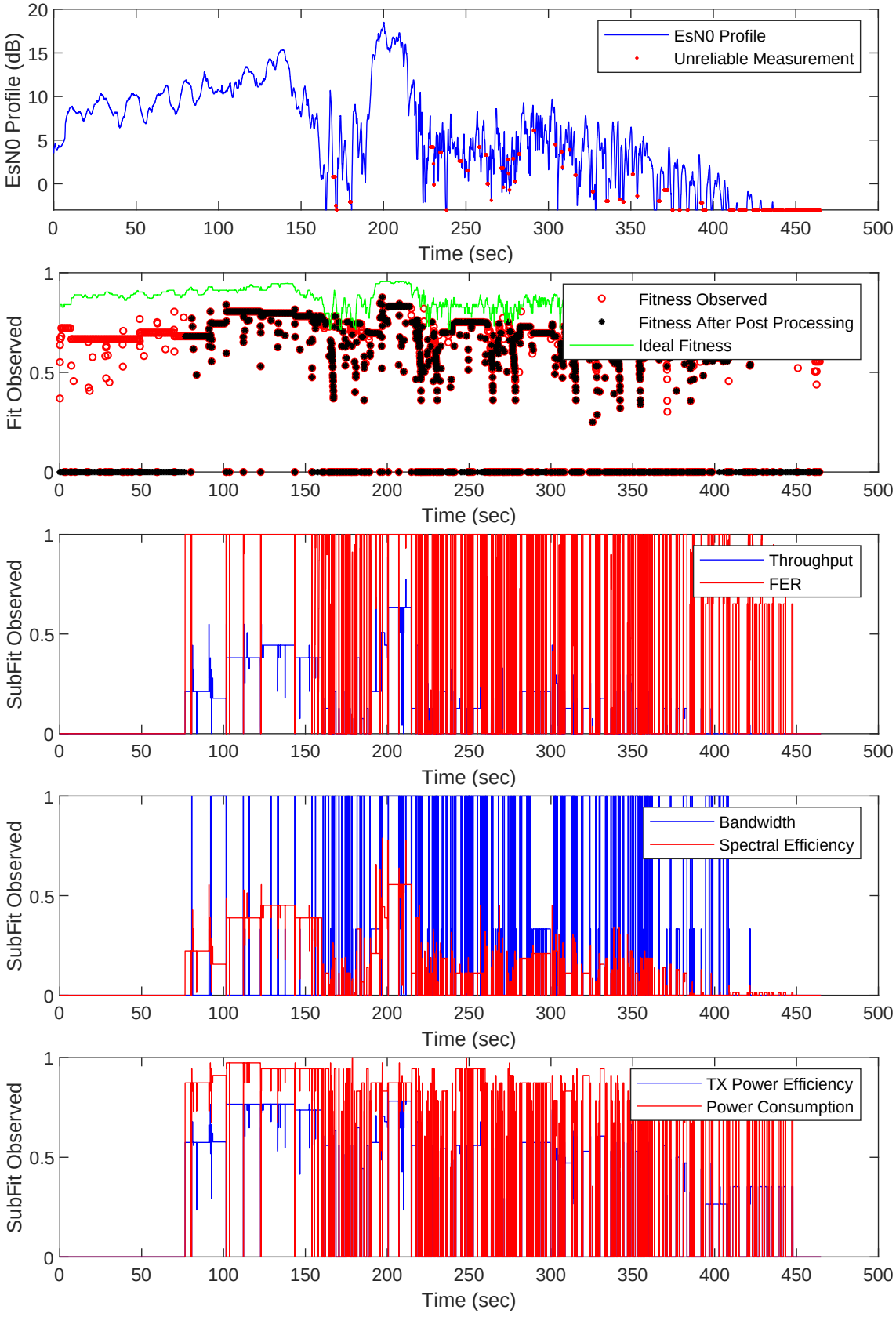


Figure A.10: Operation of CE-LM during Great quality pass using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

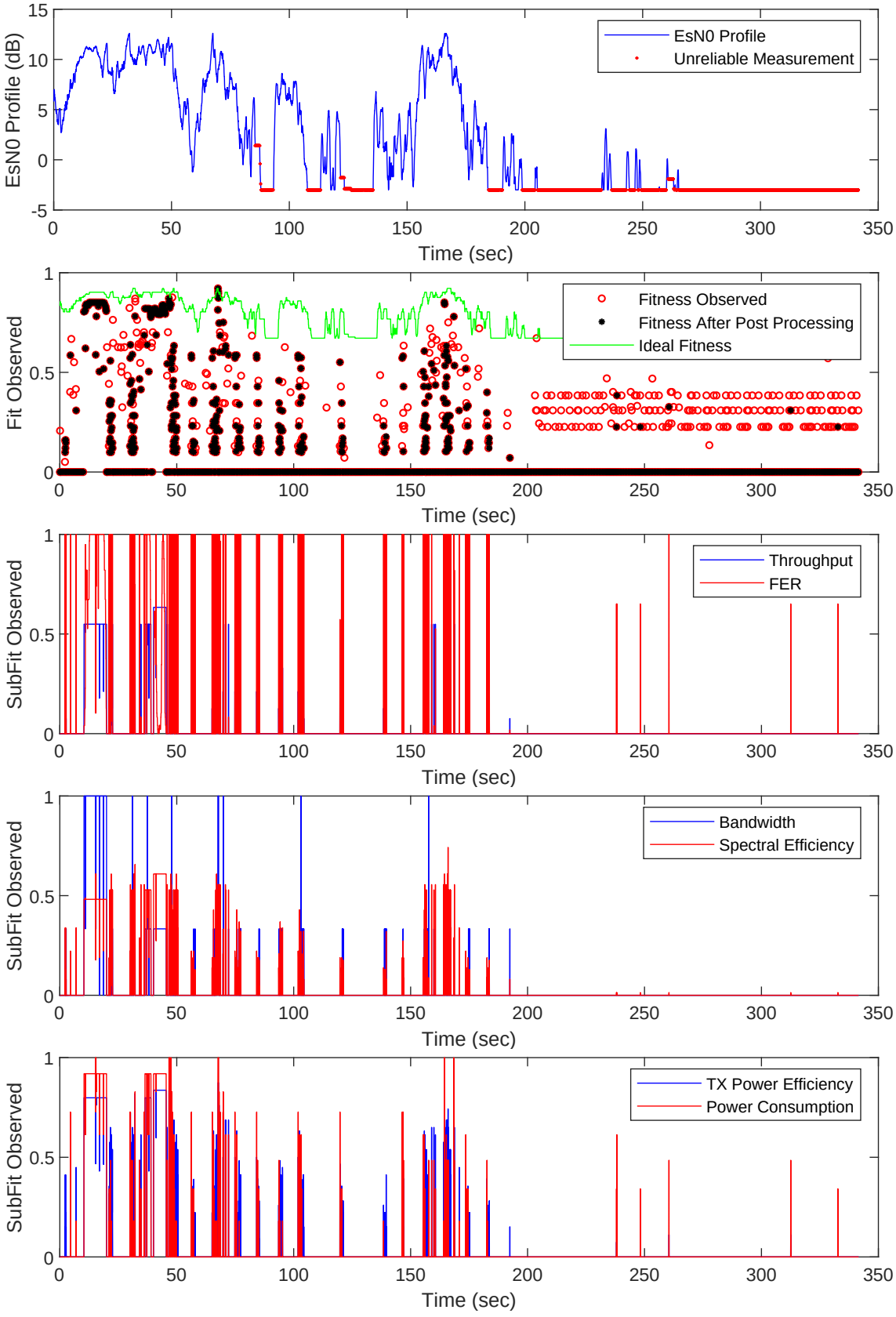


Figure A.11: Operation of CE-RLM during Great quality pass using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

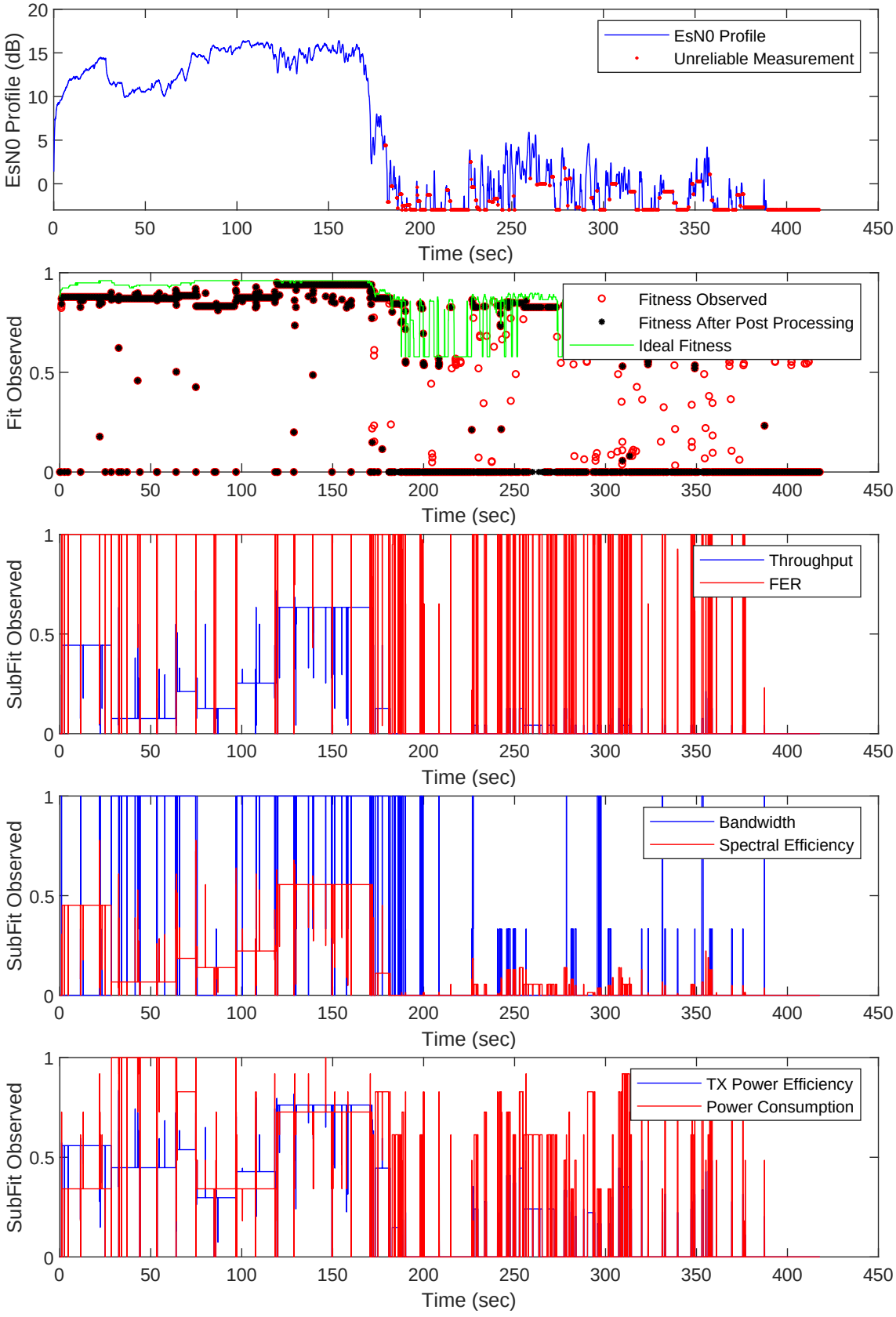


Figure A.12: Operation of CE-NSE during Great quality pass using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

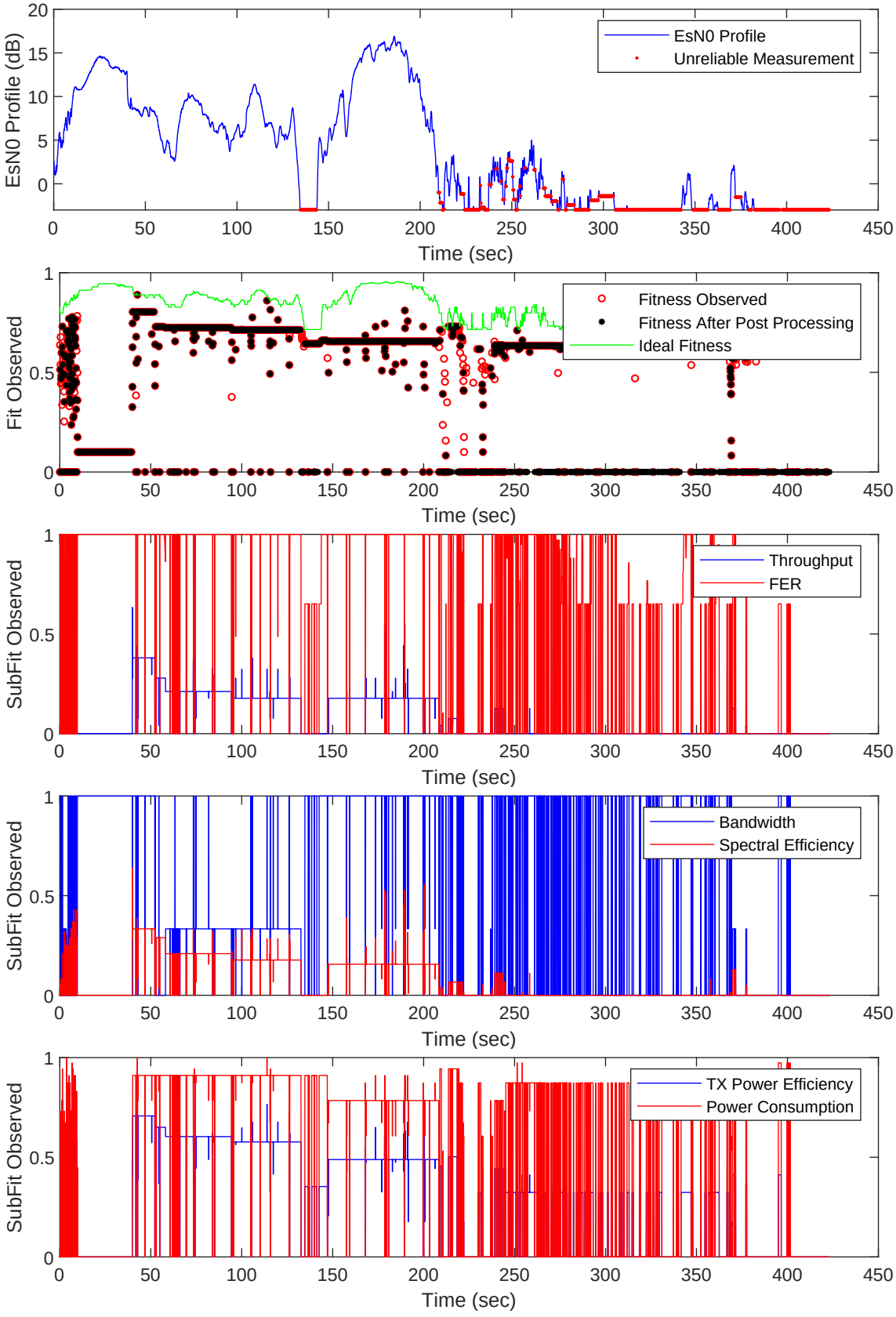


Figure A.13: Operation of CE-LM during Good quality pass using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

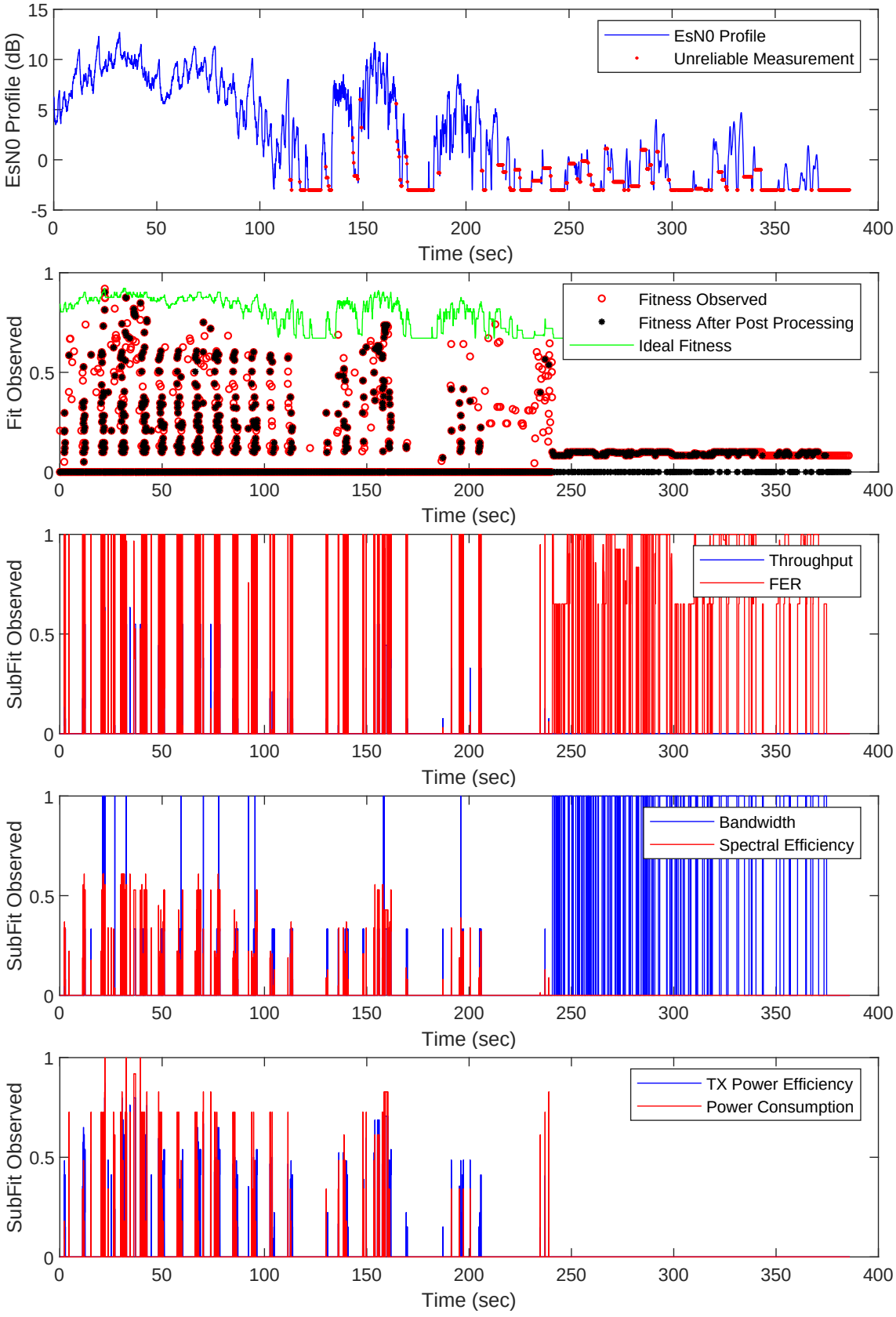


Figure A.14: Operation of CE-RLM during Good quality pass using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

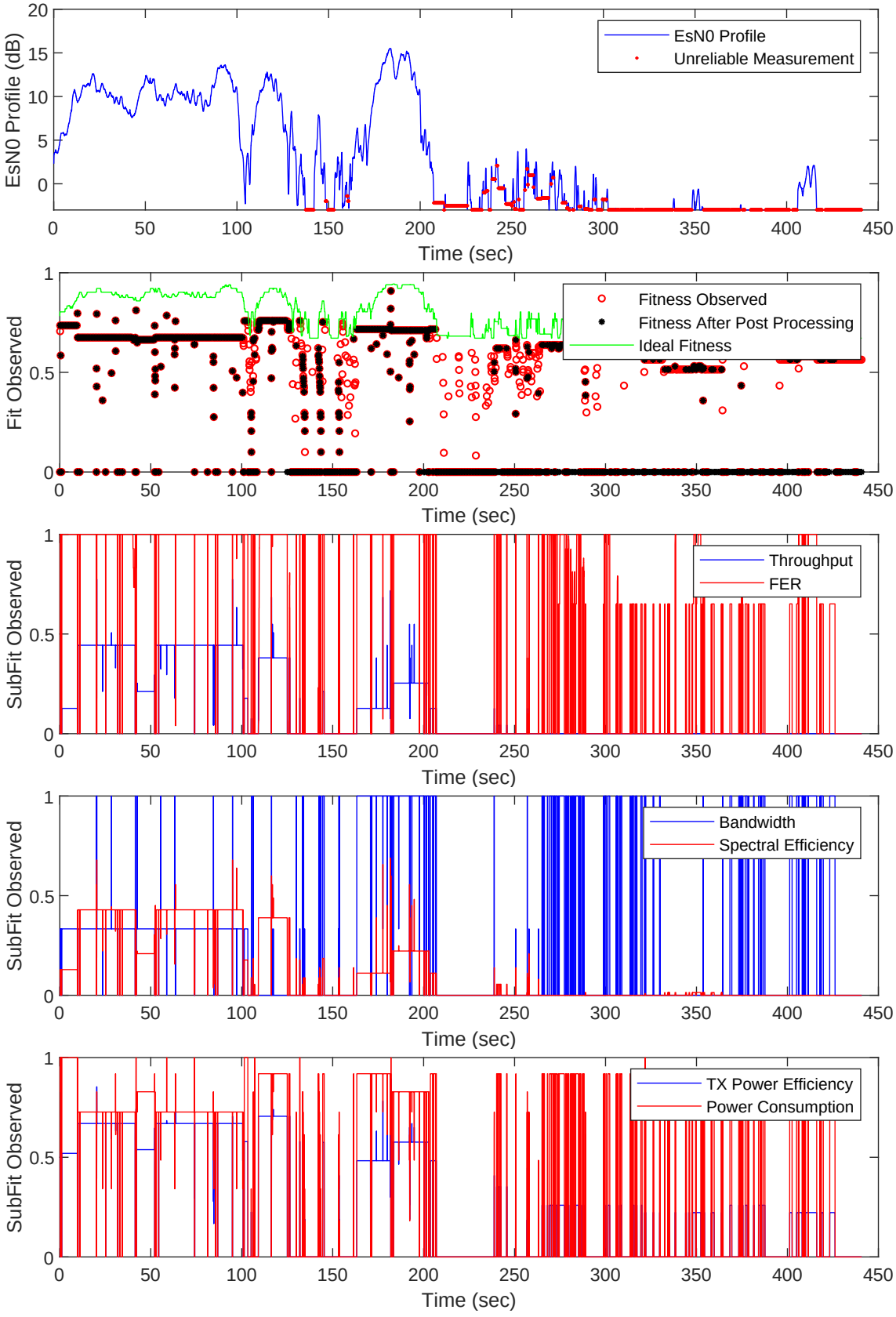


Figure A.15: Operation of CE-NSE during Good quality pass using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

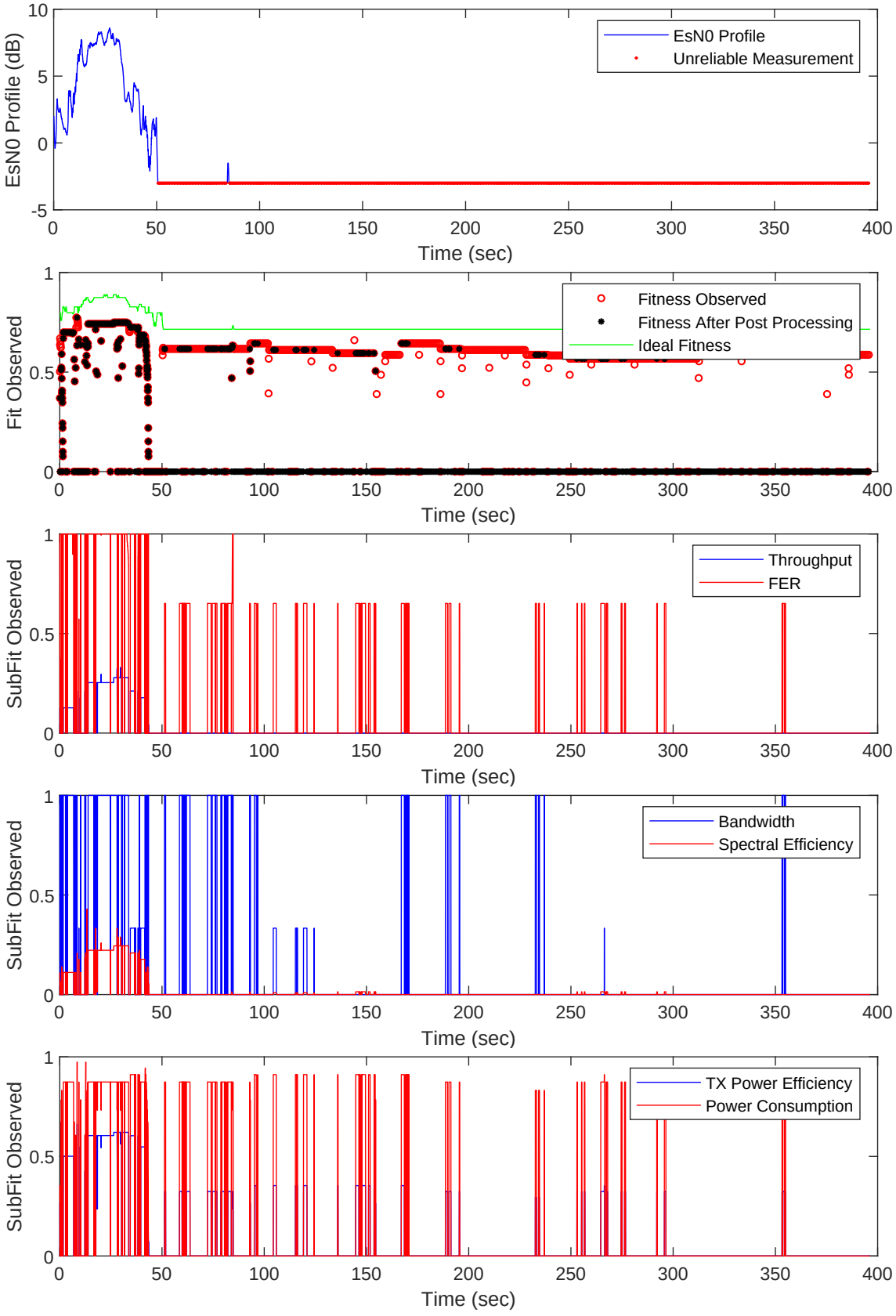


Figure A.16: Operation of CE-LM during Poor quality pass using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

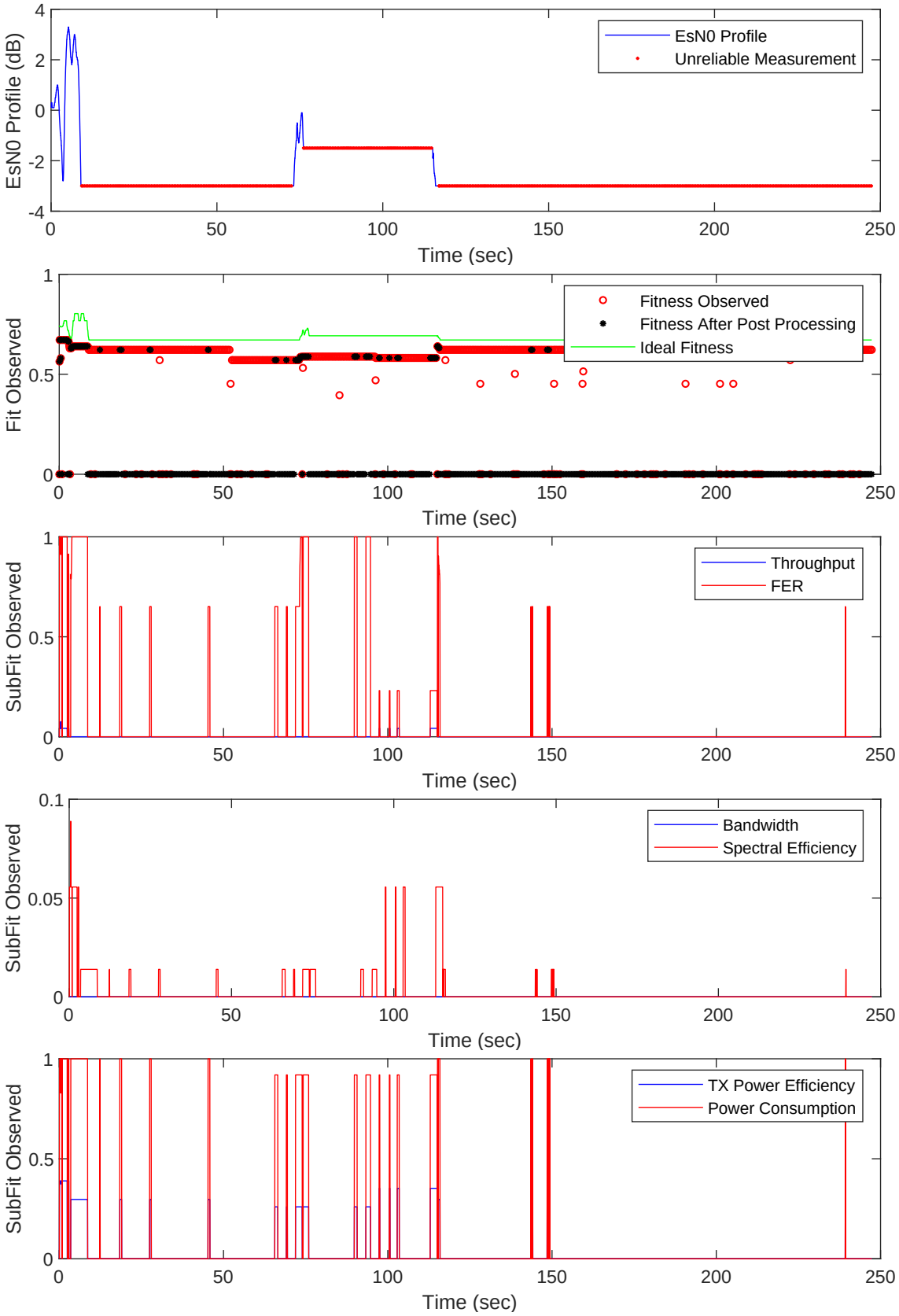


Figure A.17: Operation of CE-NSE during Poor quality pass using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

A.1.3 Emergency Mission

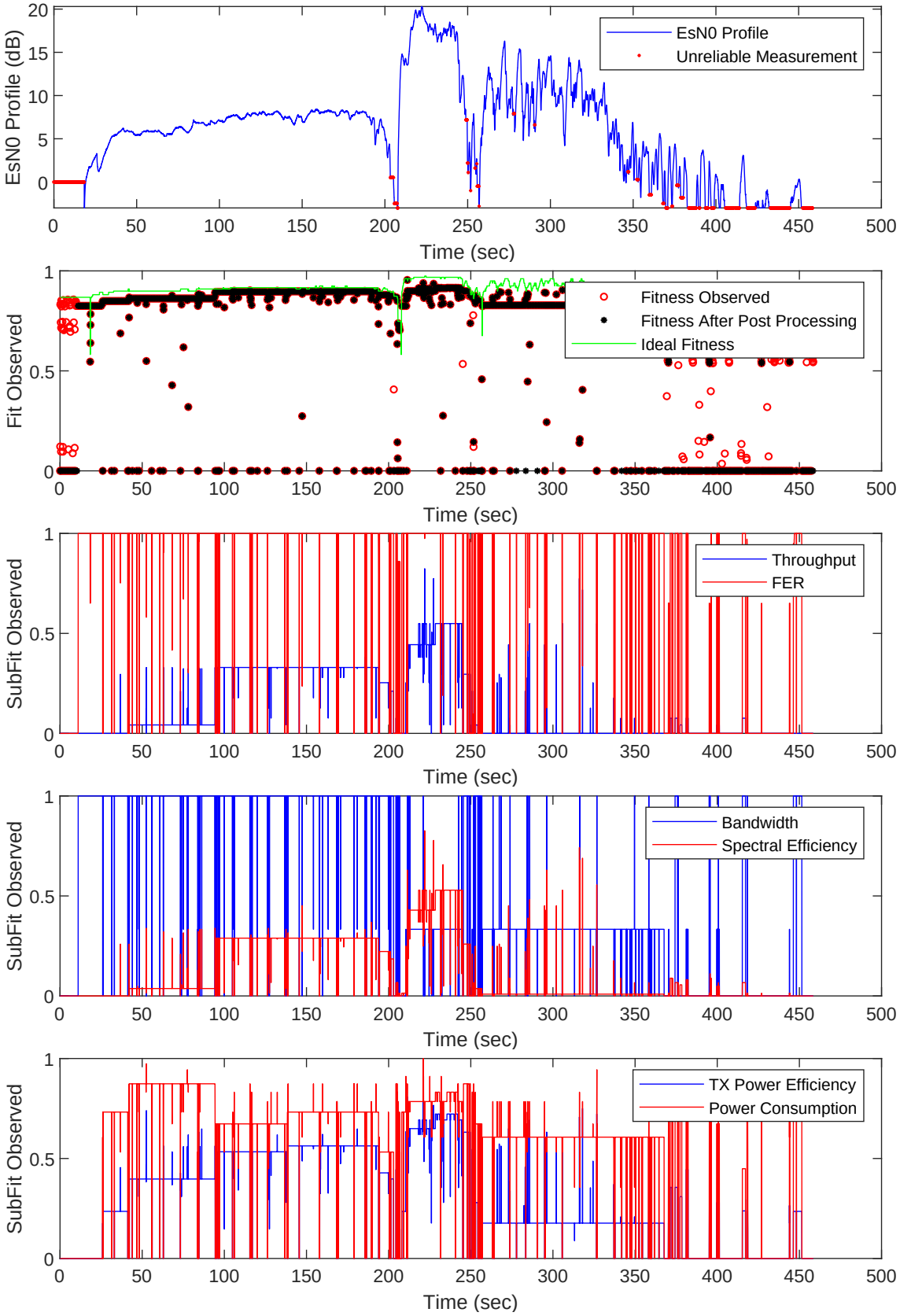


Figure A.18: Operation of CE-LM during Great quality pass using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

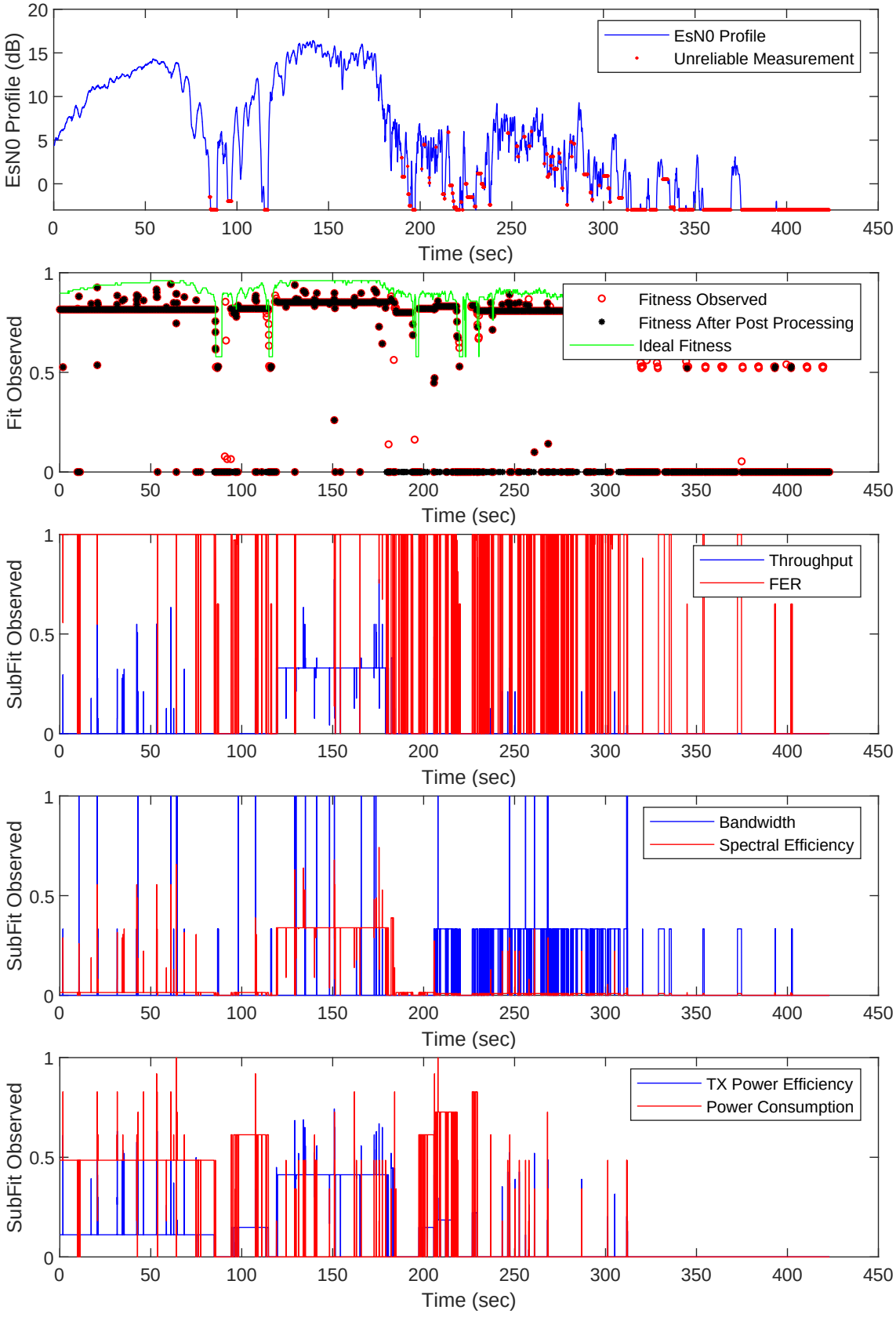


Figure A.19: Operation of CE-RLM during Great quality pass using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

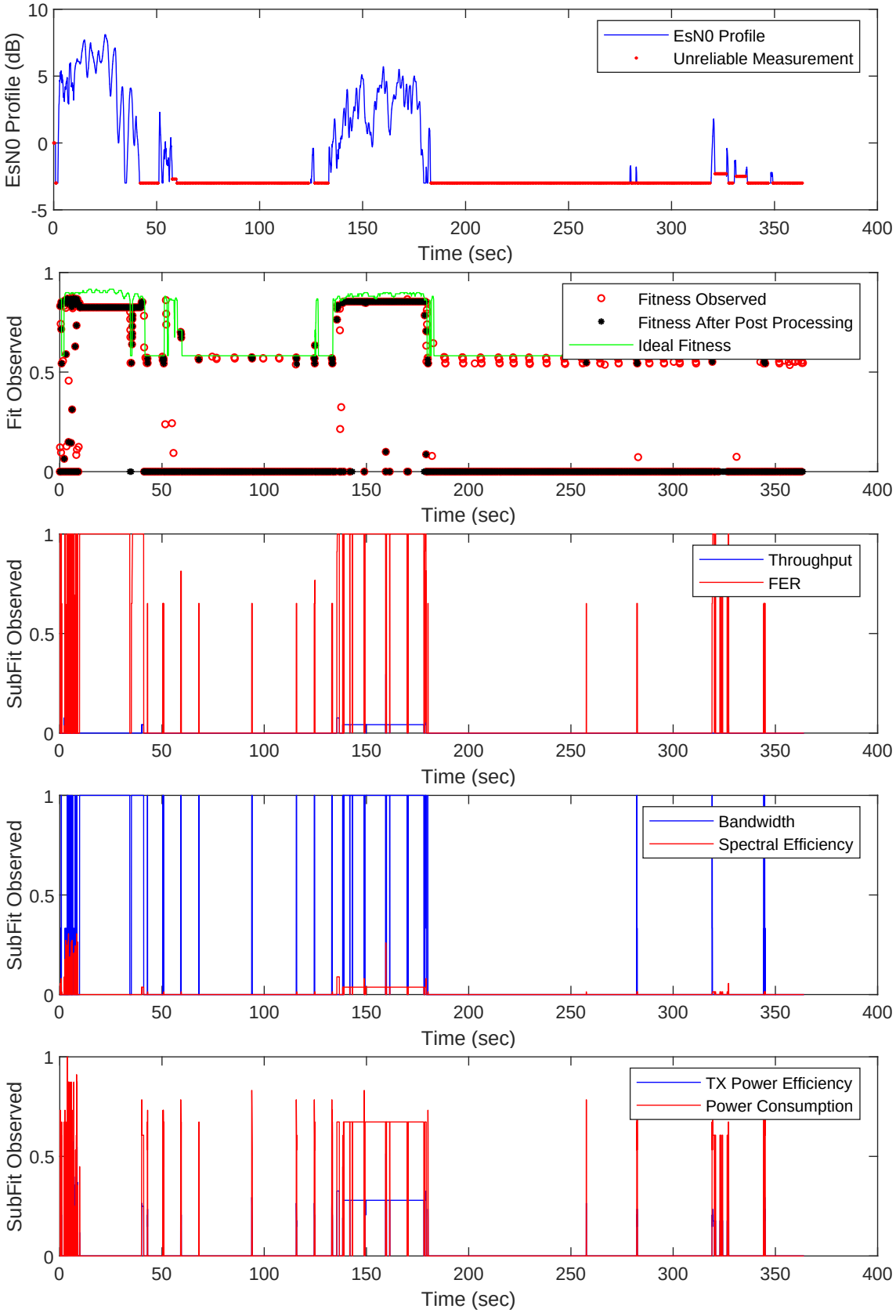


Figure A.20: Operation of CE-LM during Good quality pass using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

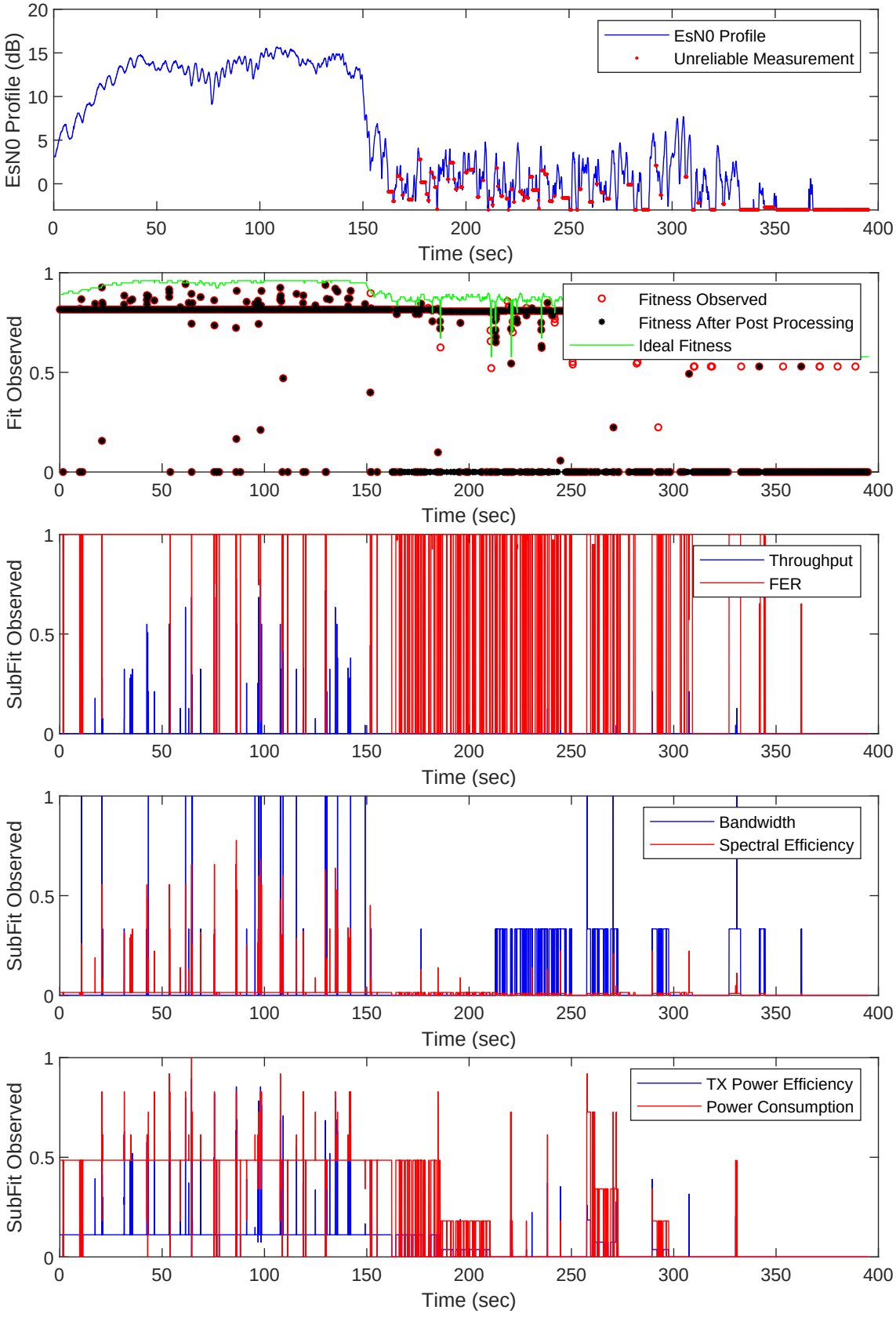


Figure A.21: Operation of CE-RLM during Good quality pass using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

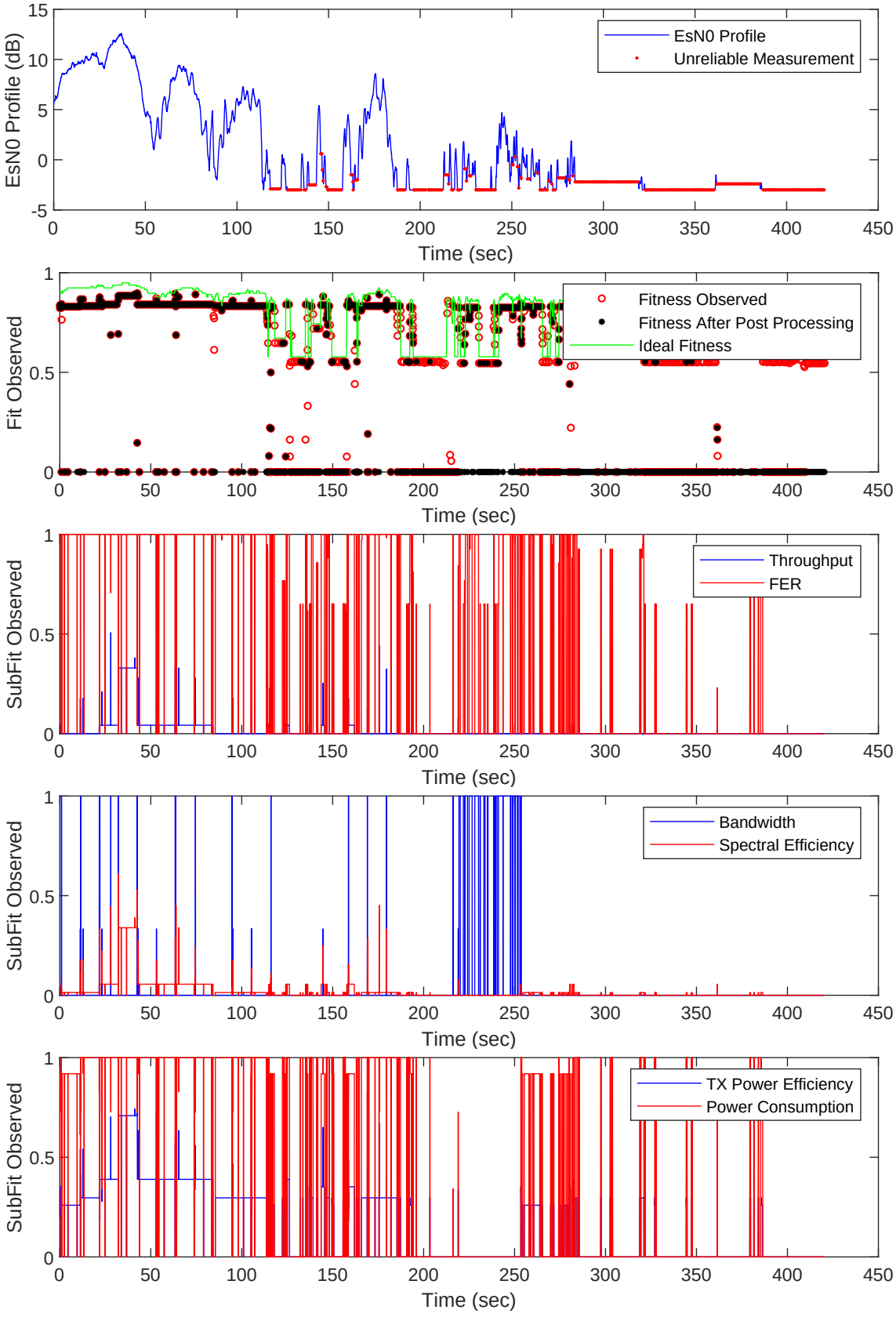


Figure A.22: Operation of CE-NSE during Good quality pass using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

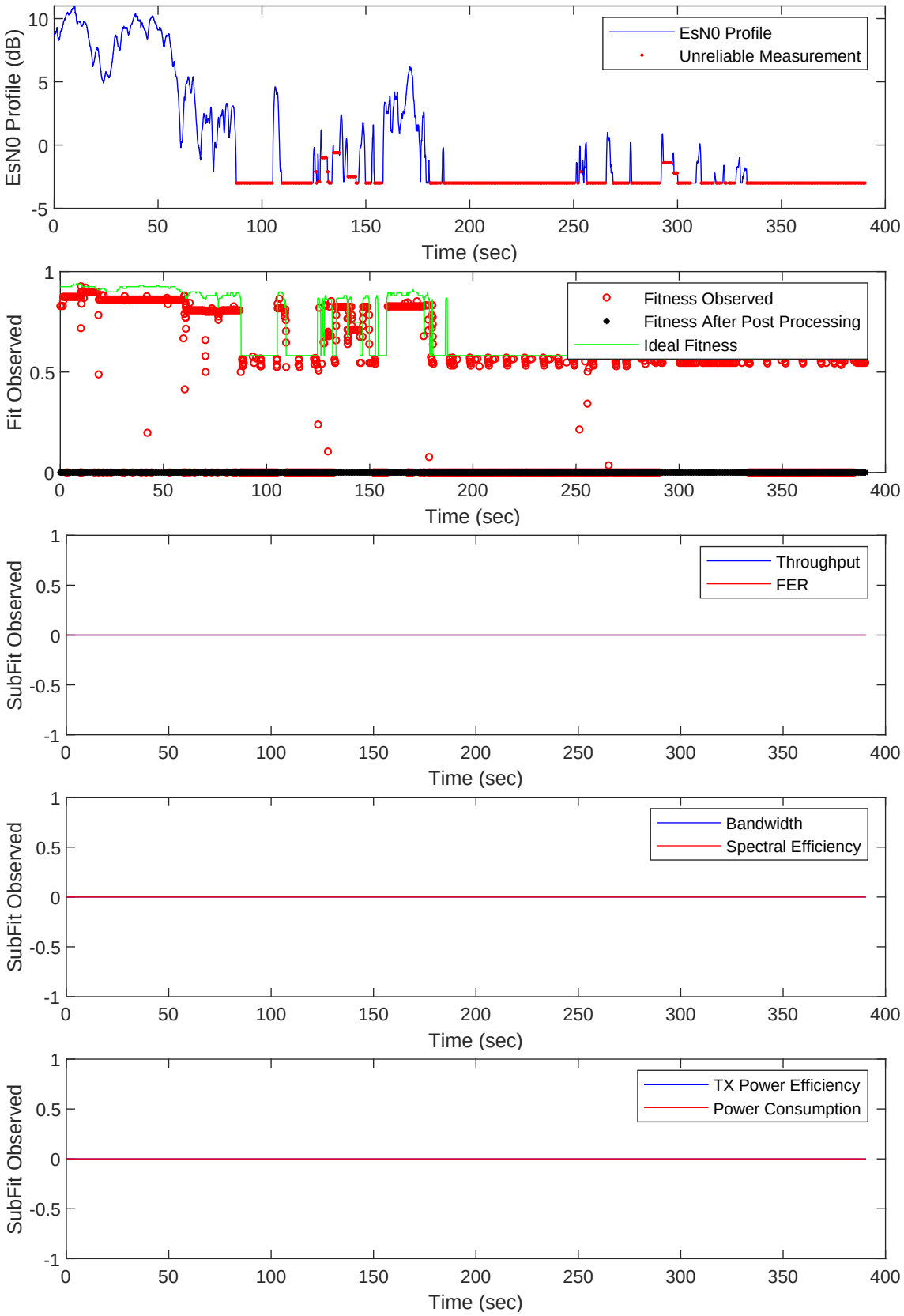


Figure A.23: Operation of CE-LM during Poor quality pass using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

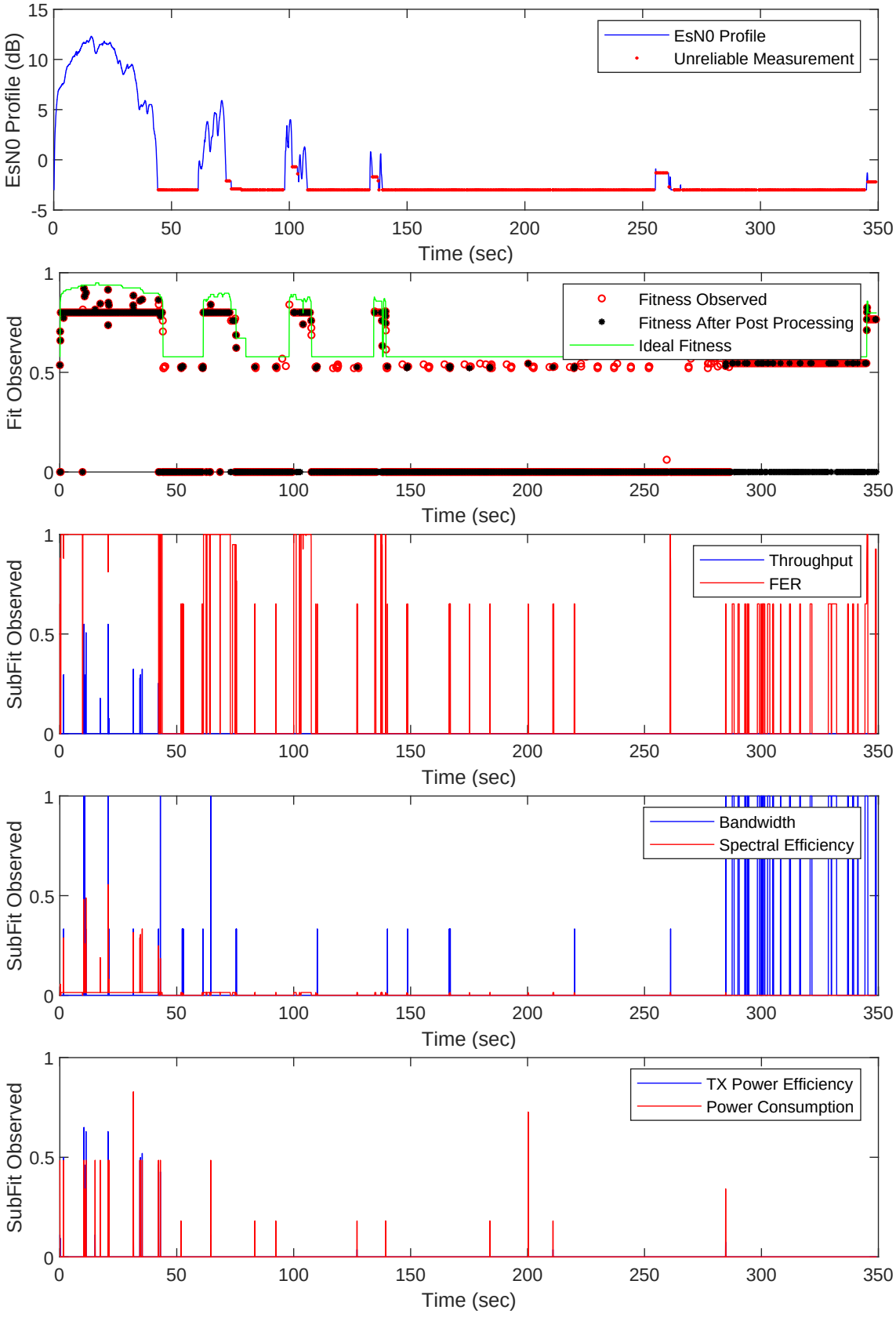


Figure A.24: Operation of CE-RLM during Poor quality pass using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

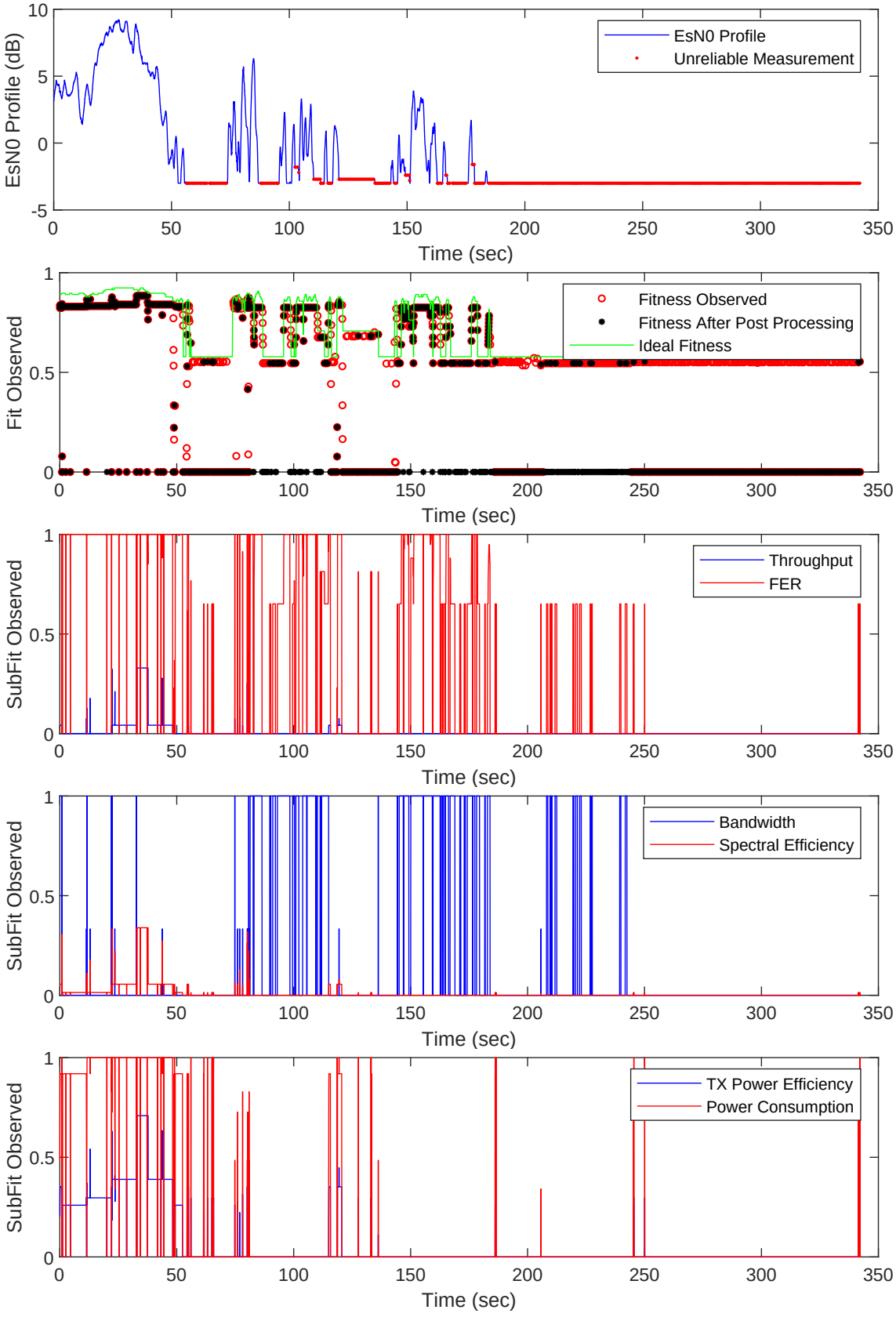
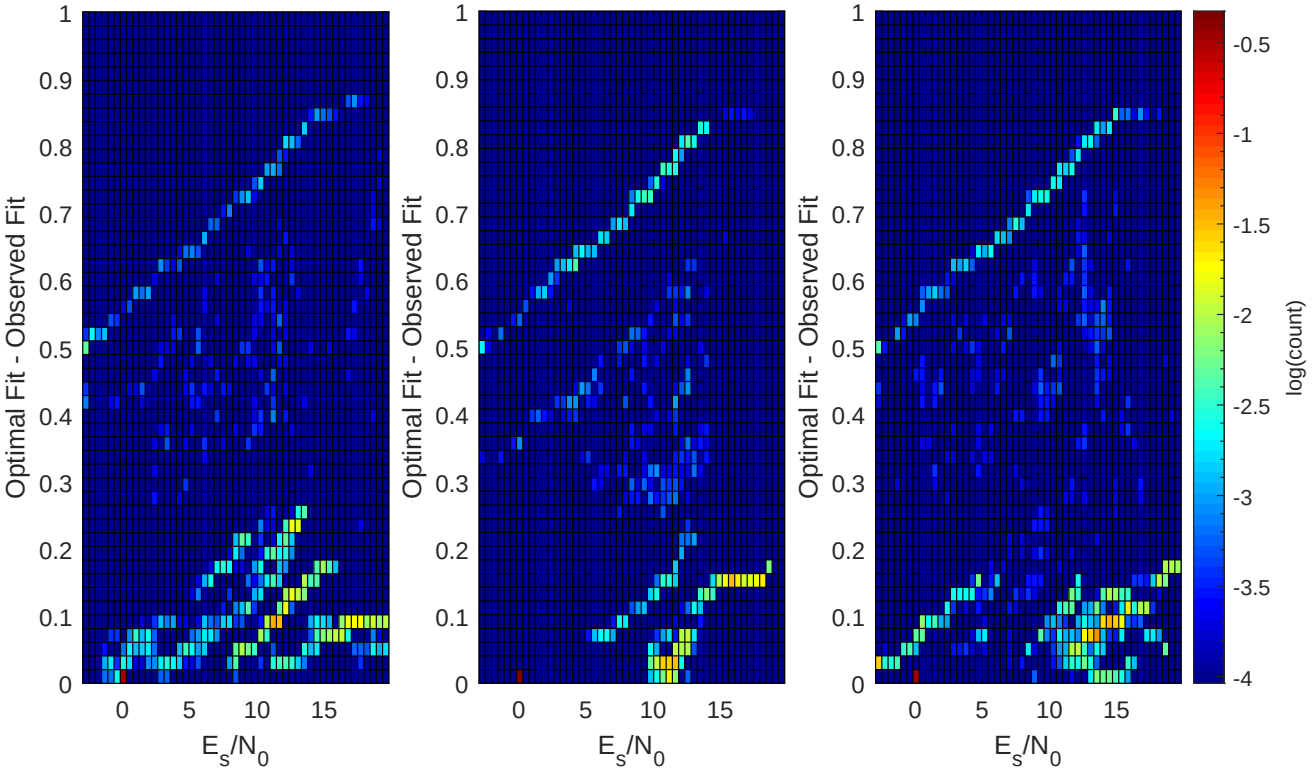


Figure A.25: Operation of CE-NSE during Poor quality pass using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

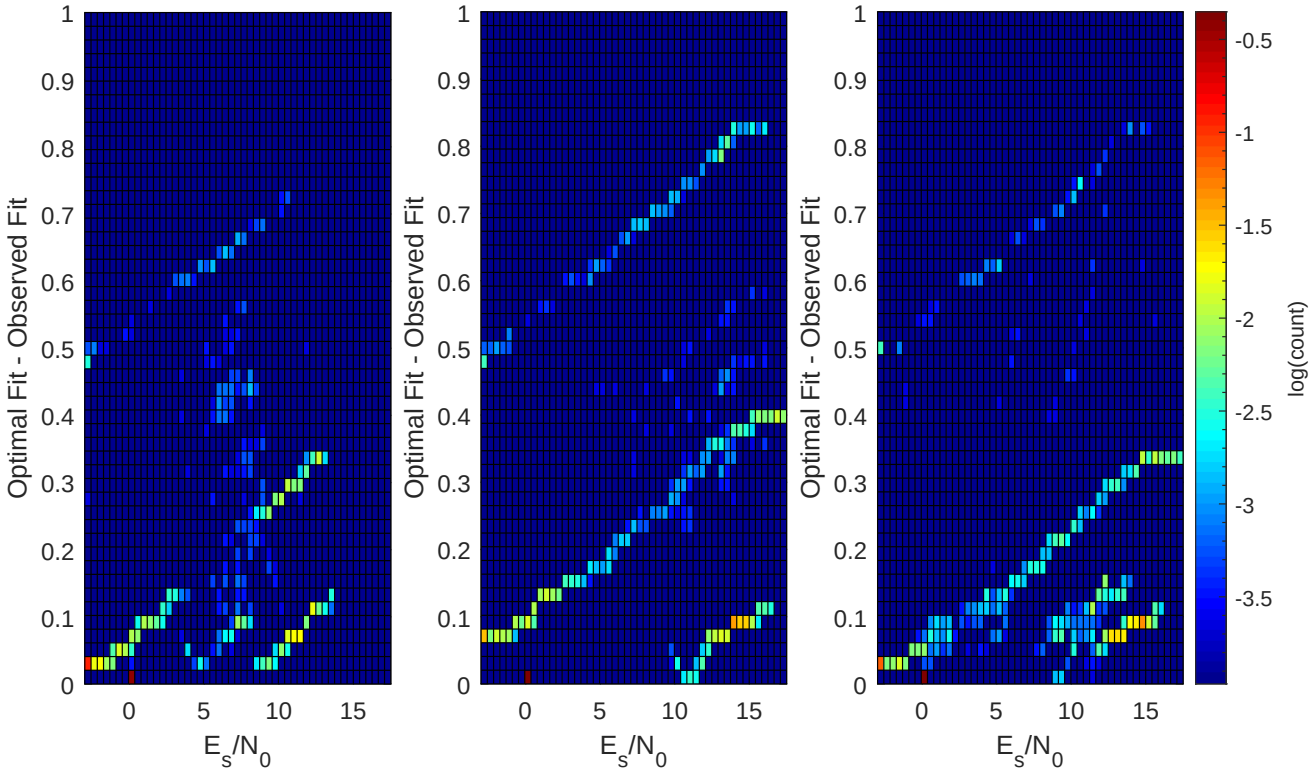
A.2 Flight Test 2D Histograms

A.2.1 Cooperation Mission



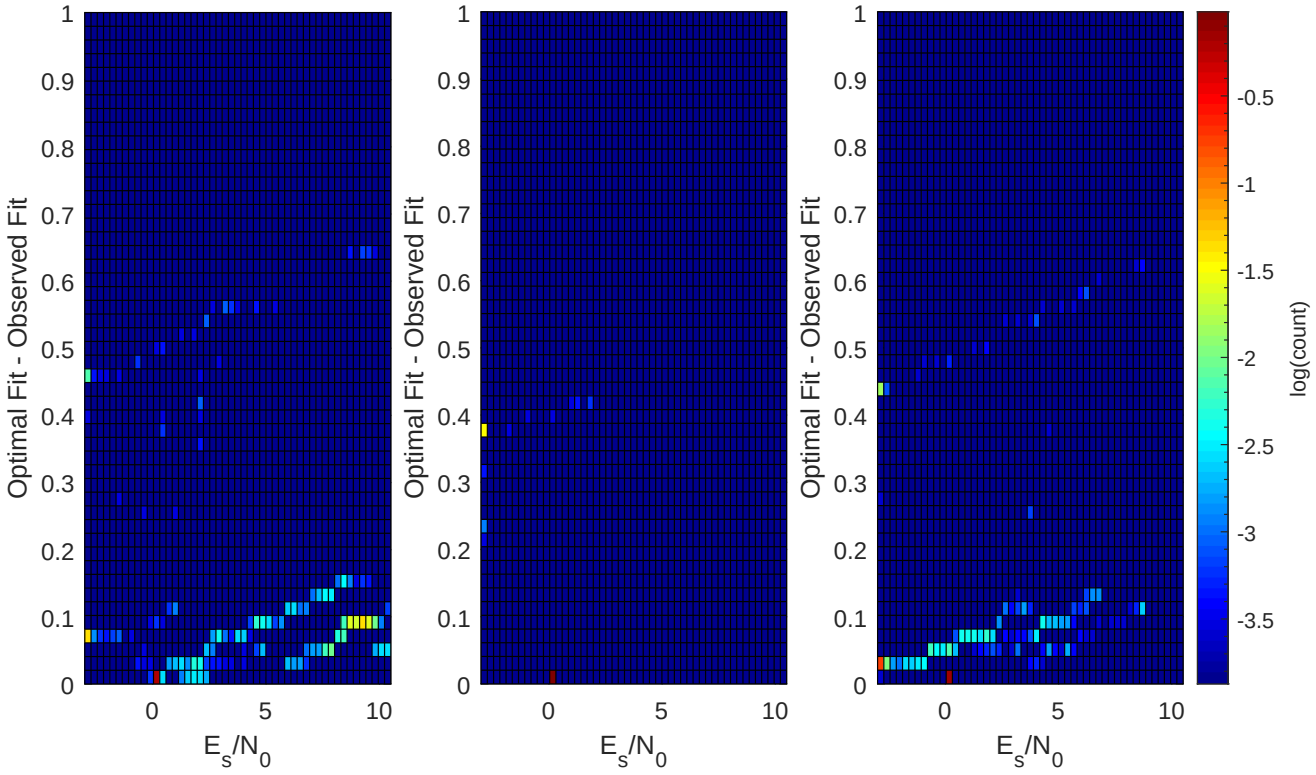
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure A.26: Two-dimensional histograms of each training method operating the Cooperation mission on a Great quality pass. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed/total number of frames)



(a) CE-LM (b) CE-RLM (c) CE-NSE

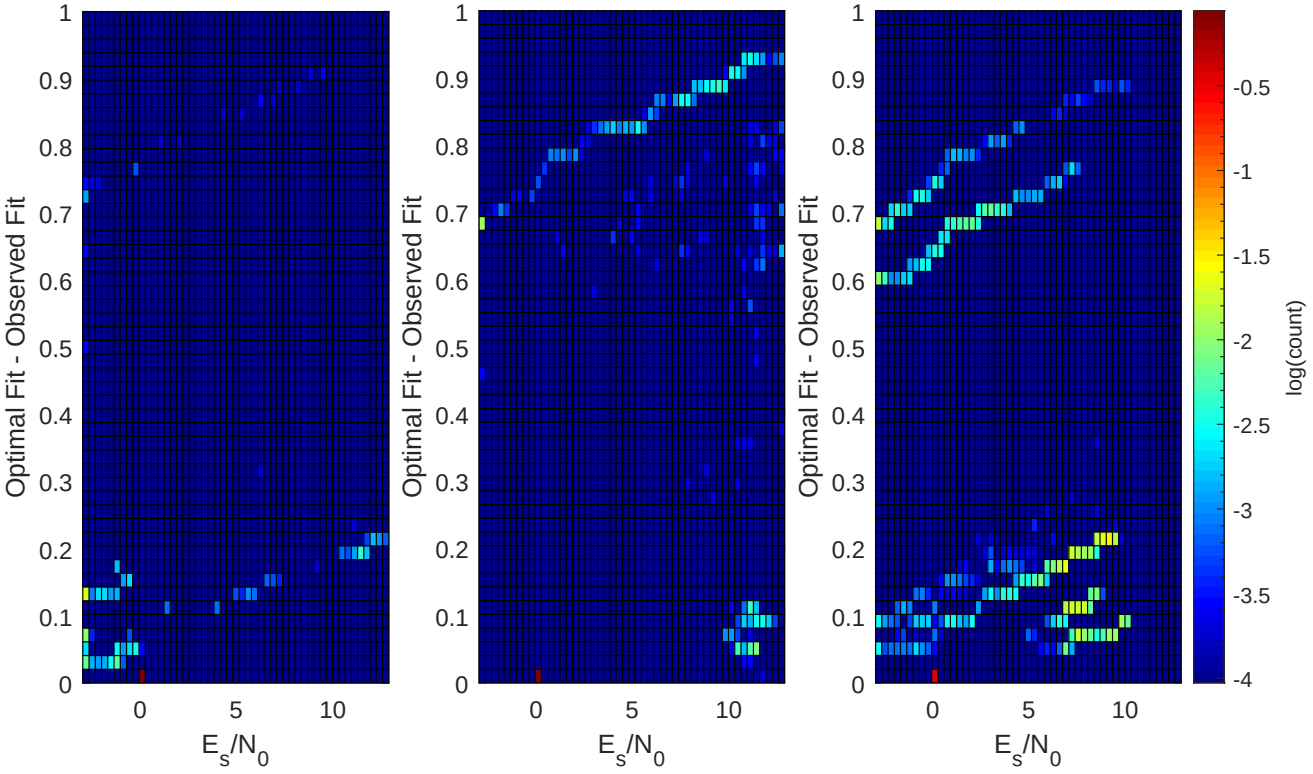
Figure A.27: Two-dimensional histograms of each training method operating the Cooperation mission on a Good quality pass. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed/total number of frames)



(a) CE-LM (b) CE-RLM (c) CE-NSE

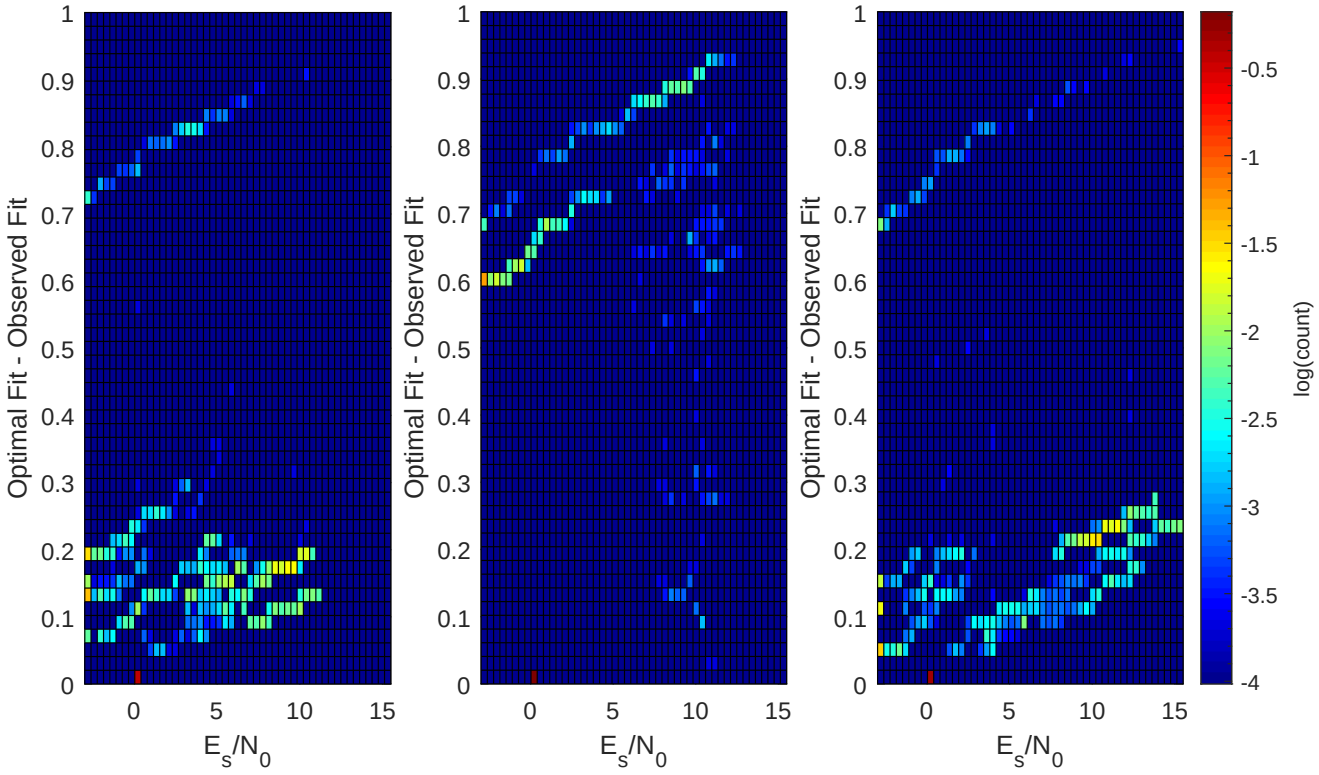
Figure A.28: Two-dimensional histograms of each training method operating the Cooperation mission on a Poor quality pass. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed/total number of frames)

A.2.2 Power Saving Mission



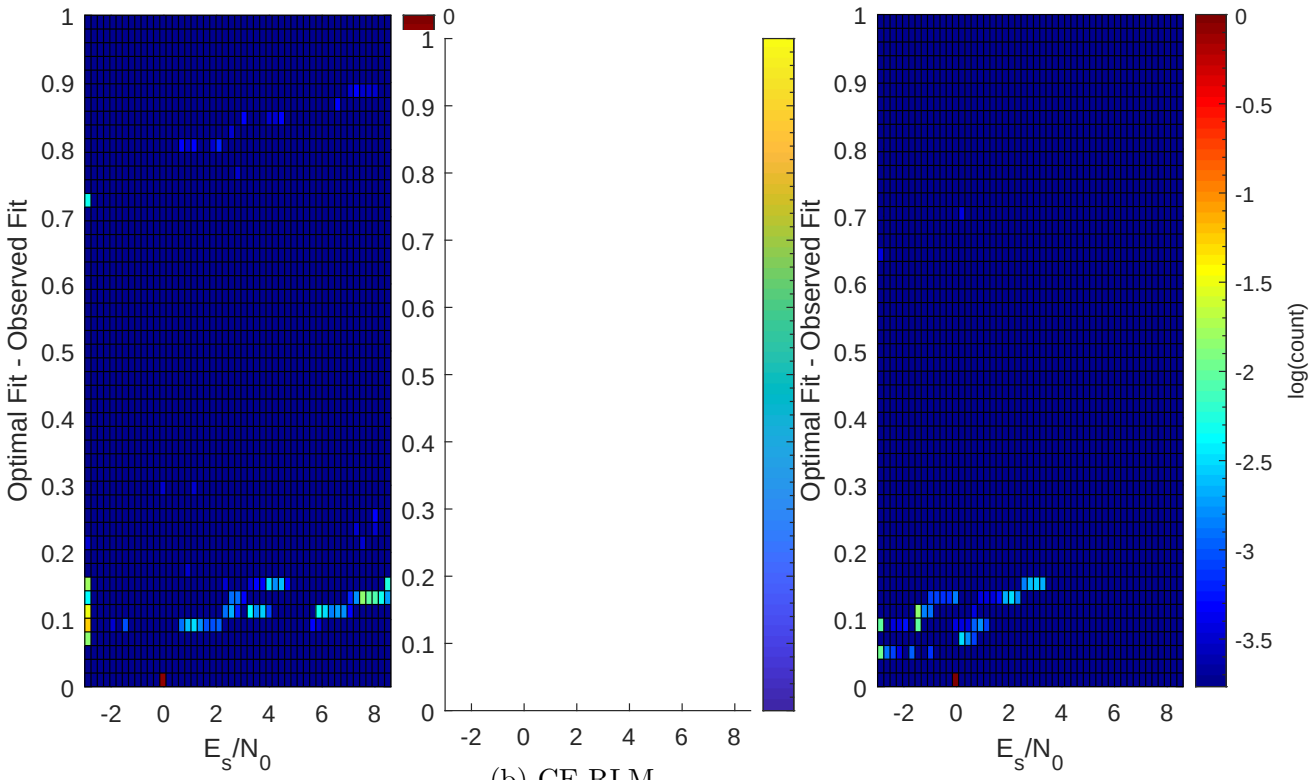
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure A.29: Two-dimensional histograms of each training method operating the Power Saving mission on a Great quality pass. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed/total number of frames)



(a) CE-LM (b) CE-RLM (c) CE-NSE

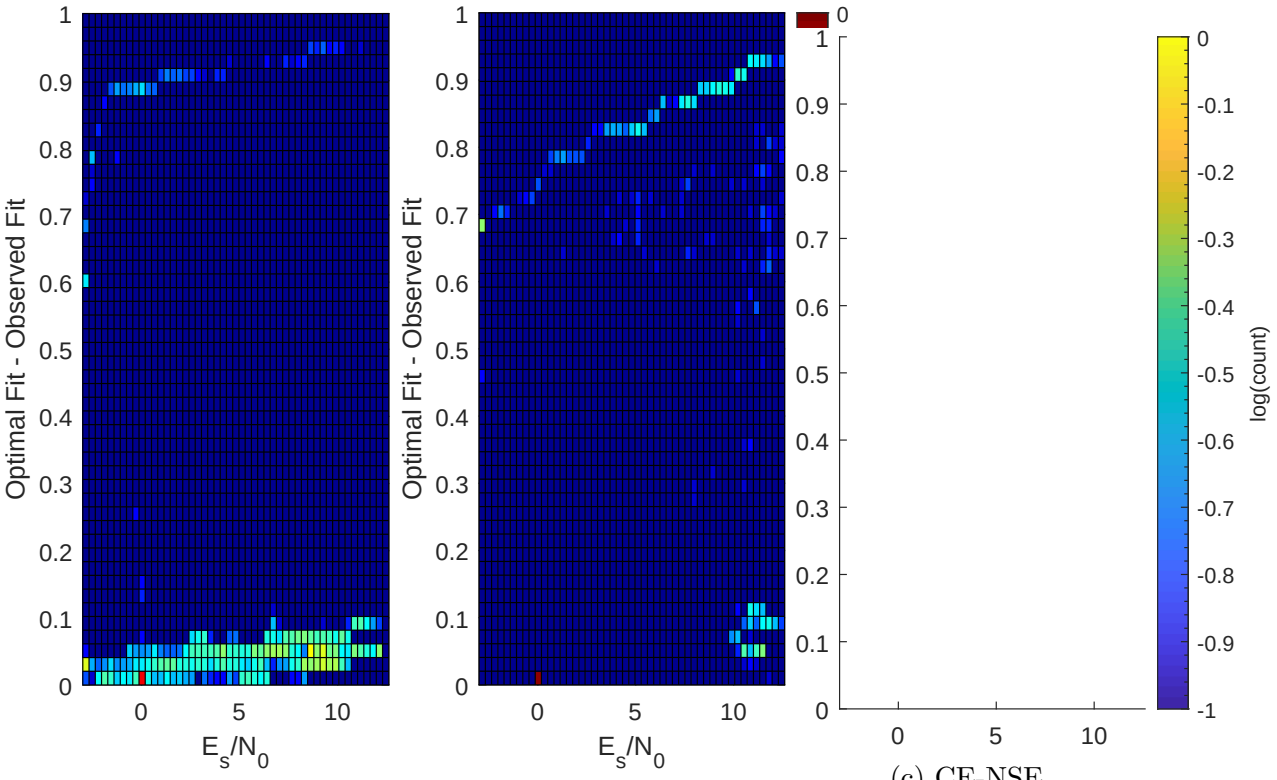
Figure A.30: Two-dimensional histograms of each training method operating the Power Saving mission on a Good quality pass. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed/total number of frames)



(a) CE-LM (b) CE-RLM (c) CE-NSE

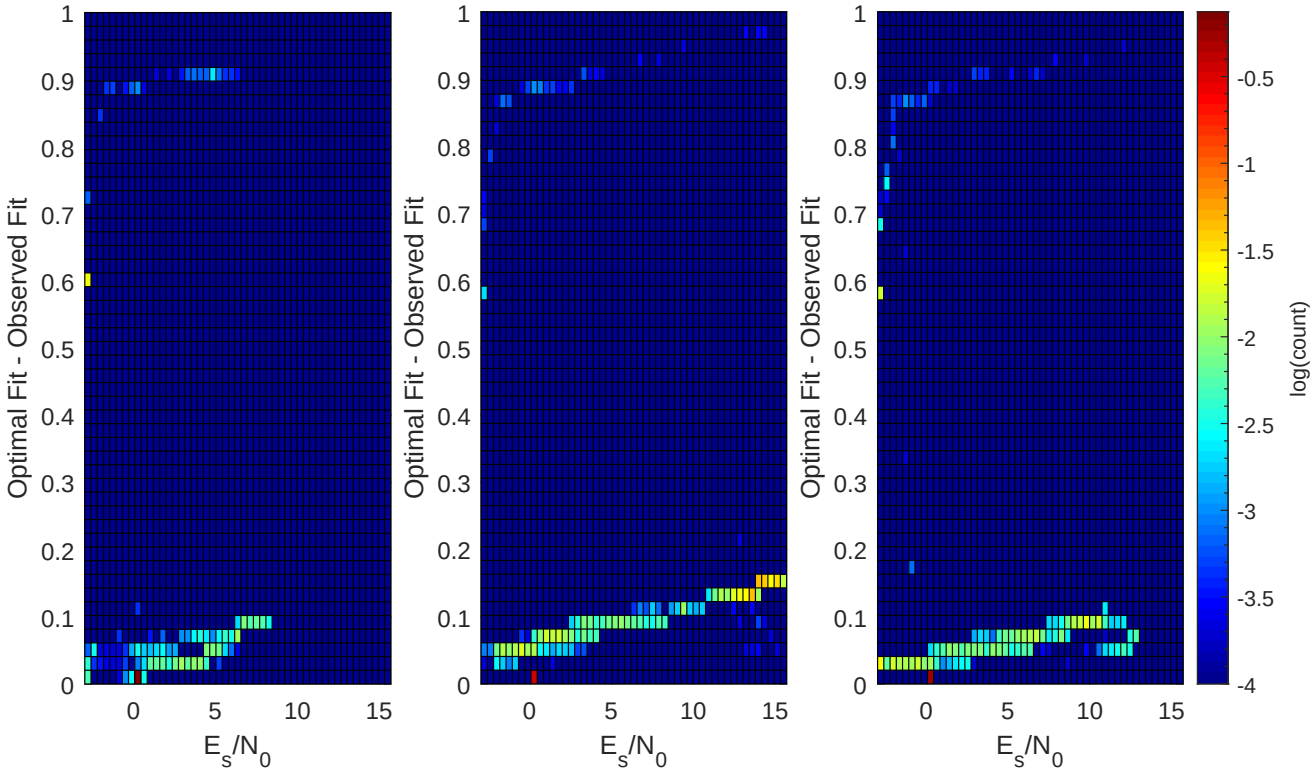
Figure A.31: Two-dimensional histograms of each training method operating the Power Saving mission on a Poor quality pass. During the pass that RLM was operating, the modems never locked on. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed/total number of frames)

A.2.3 Emergency Mission



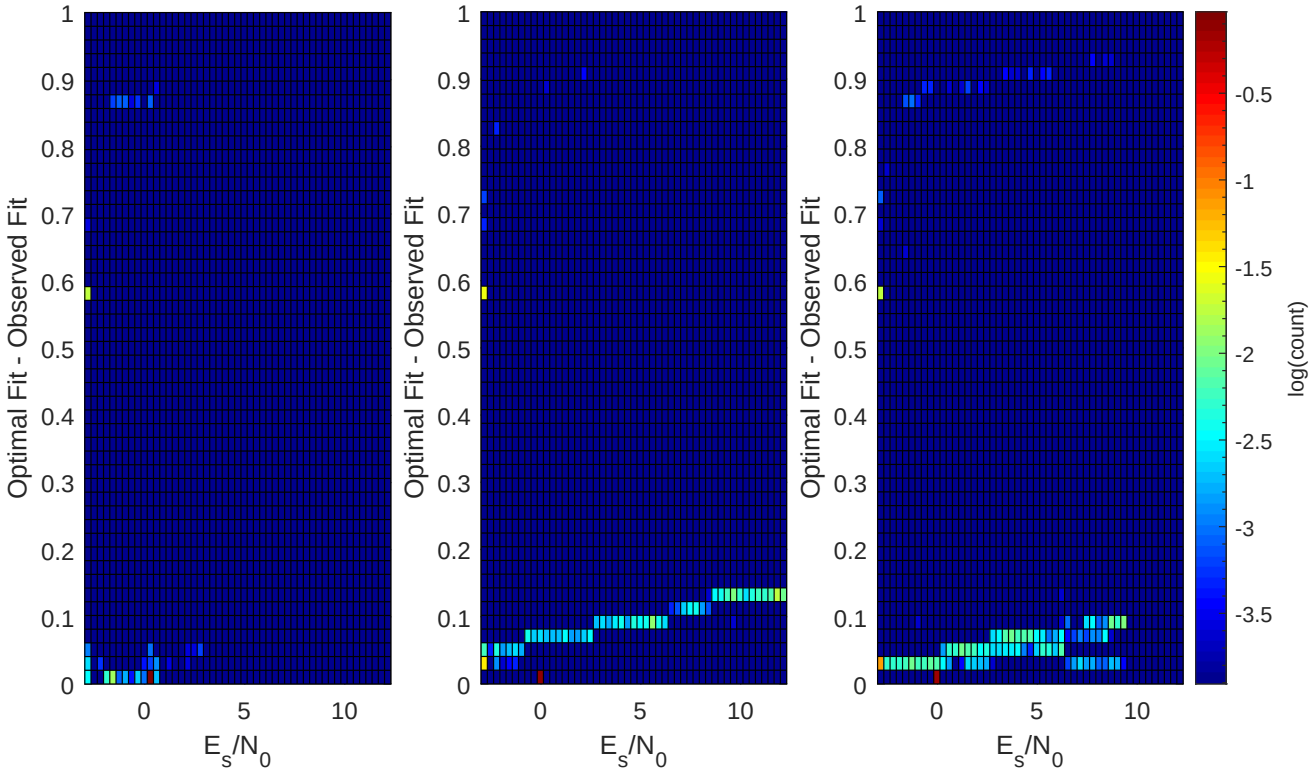
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure A.32: Two-dimensional histograms of each training method operating the Emergency mission on a Great quality pass. A Great quality pass was not recorded for CE-NSE operating the Emergency mission. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed/total number of frames)



(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure A.33: Two-dimensional histograms of each training method operating the Emergency mission on a Good quality pass. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed/total number of frames)

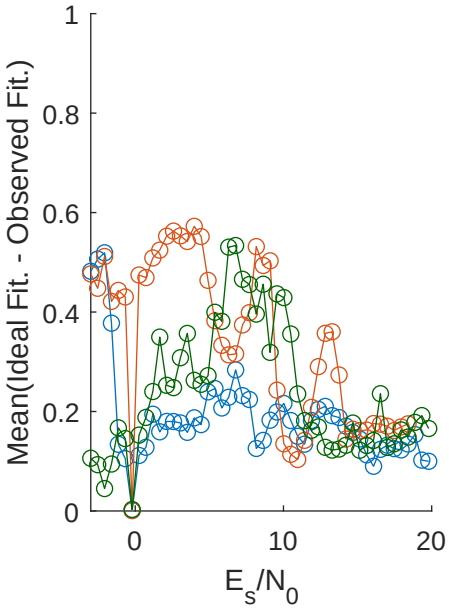


(a) CE-LM (b) CE-RLM (c) CE-NSE

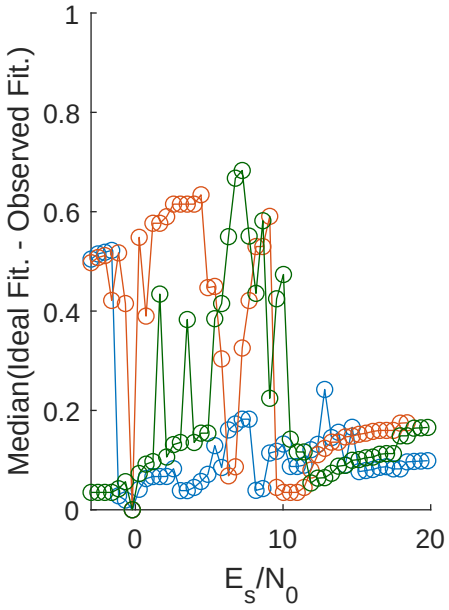
Figure A.34: Two-dimensional histograms of each training method operating the Emergency mission on a Poor quality pass. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed/total number of frames)

A.3 Flight Test Binned Value Plots

A.3.1 Cooperation Mission

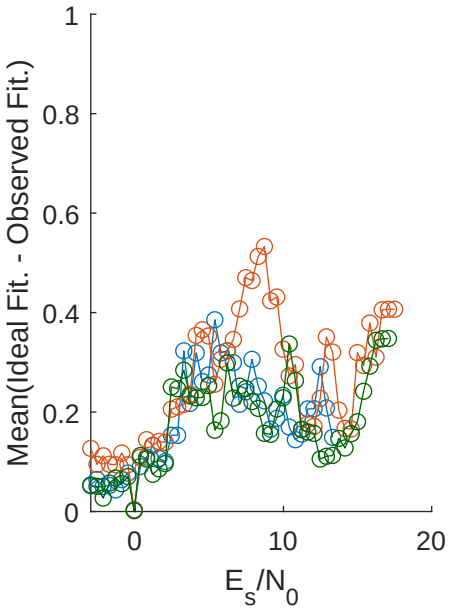


(a) Binned mean for all CE variants during a Great quality pass using Cooperation mission.

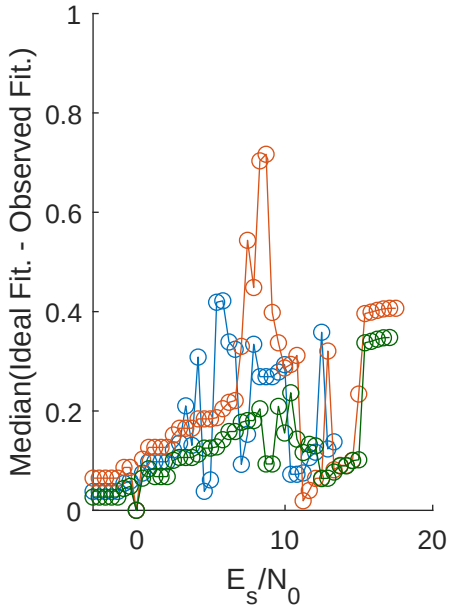


(b) Binned median for all CE variants during a Great quality pass using Cooperation mission.

Figure A.35: Binned mean and median plots for all CE variants, using Cooperation mission over a Great quality pass.

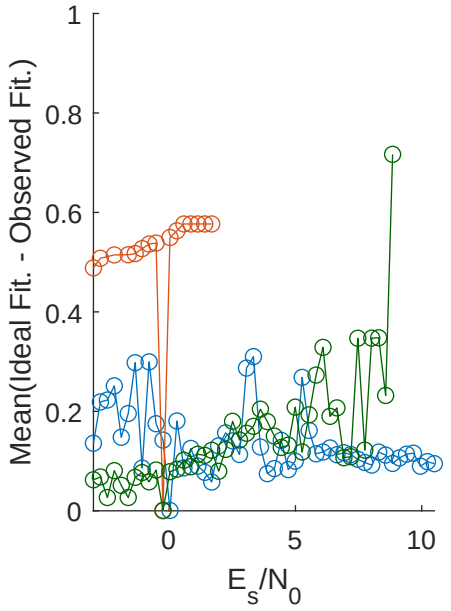


(a) Binned mean for all CE variants during a Good quality pass using Cooperation mission.

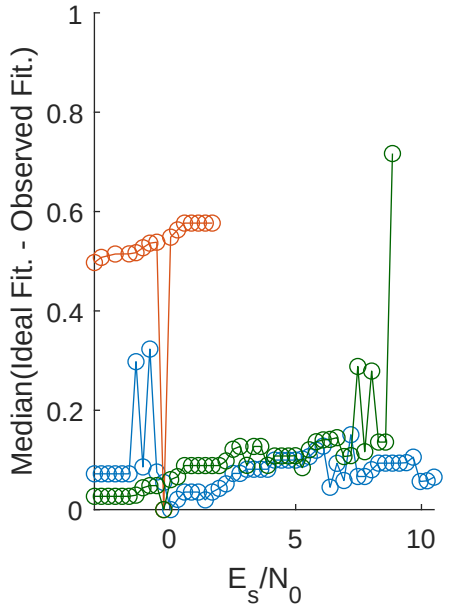


(b) Binned median for all CE variants during a Good quality pass using Cooperation mission.

Figure A.36: Binned mean and median plots for all CE variants, using Cooperation mission over a Good quality pass.



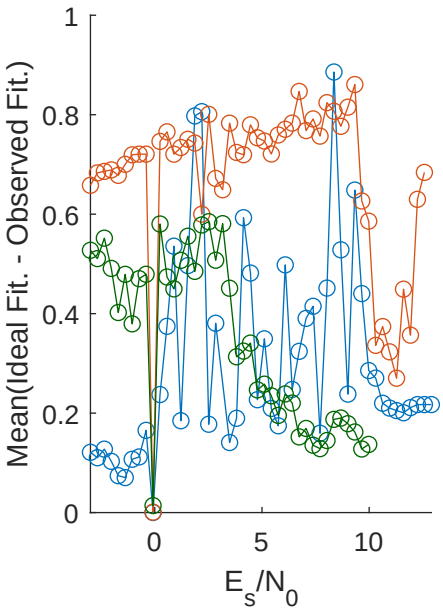
(a) Binned mean for all CE variants during a Poor quality pass using Cooperation mission.



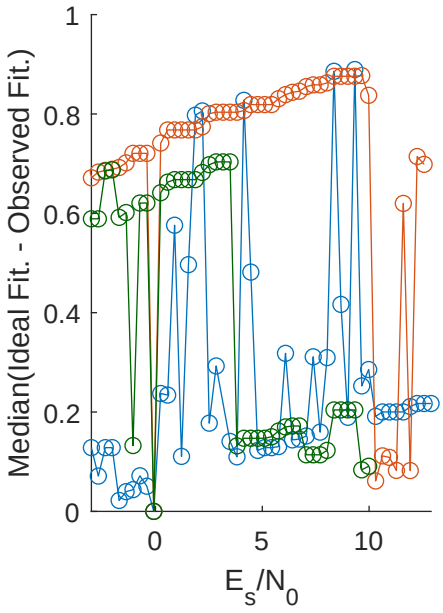
(b) Binned median for all CE variants during a Poor quality pass using Cooperation mission.

Figure A.37: Binned mean and median plots for all CE variants, using Cooperation mission over a Poor quality pass.

A.3.2 Power Saving Mission

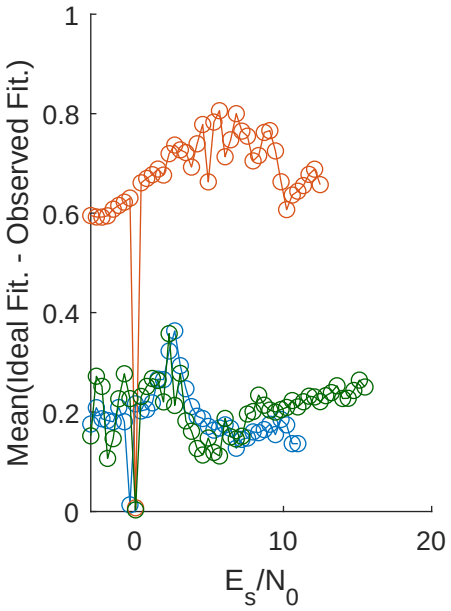


(a) Binned mean for all CE variants during a Great quality pass using Power Saving mission.

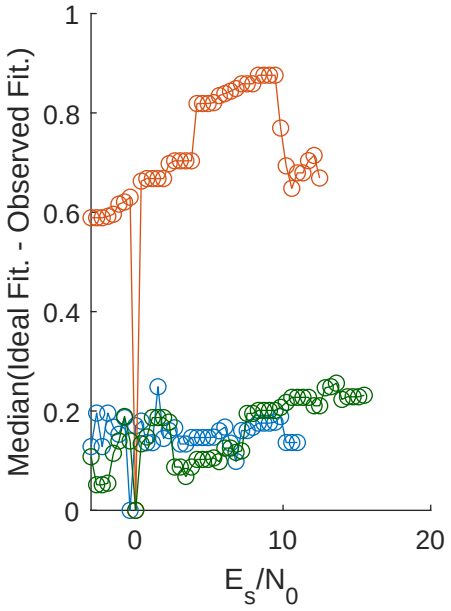


(b) Binned median for all CE variants during a Great quality pass using Power Saving mission.

Figure A.38: Binned mean and median plots for all CE variants, using Power Saving mission over a Great quality pass.

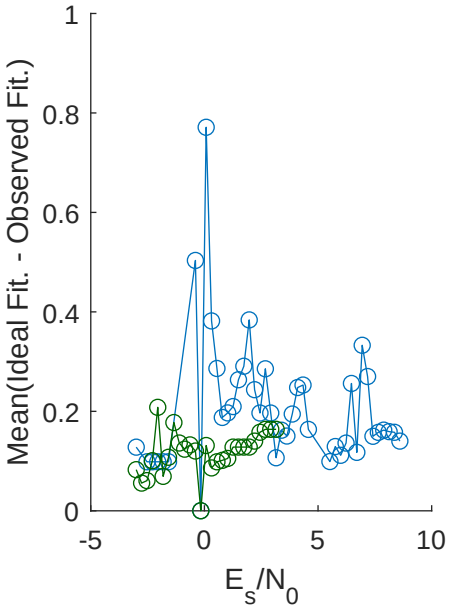


(a) Binned mean for all CE variants during a Good quality pass using Power Saving mission.

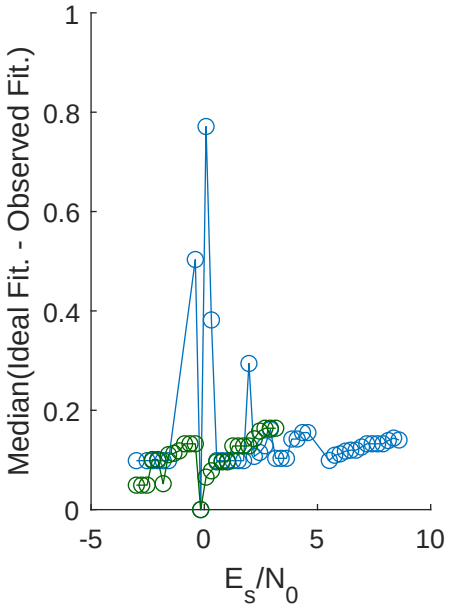


(b) Binned median for all CE variants during a Good quality pass using Power Saving mission.

Figure A.39: Binned mean and median plots for all CE variants, using Power Saving mission over a Good quality pass.



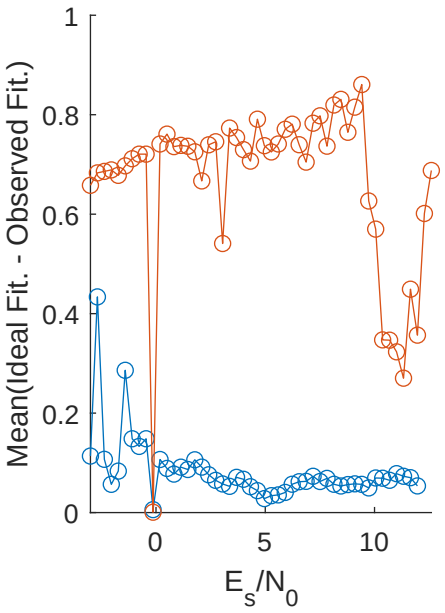
(a) Binned mean for all CE variants during a Poor quality pass using Power Saving mission.



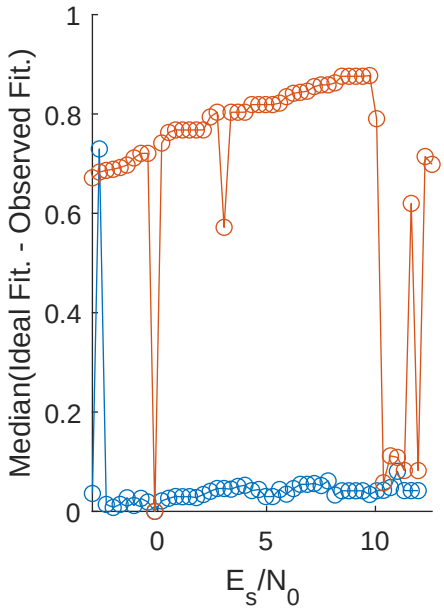
(b) Binned median for all CE variants during a Poor quality pass using Power Saving mission.

Figure A.40: Binned mean and median plots for all CE variants, using Power Saving mission over a Poor quality pass.

A.3.3 Cooperation Mission

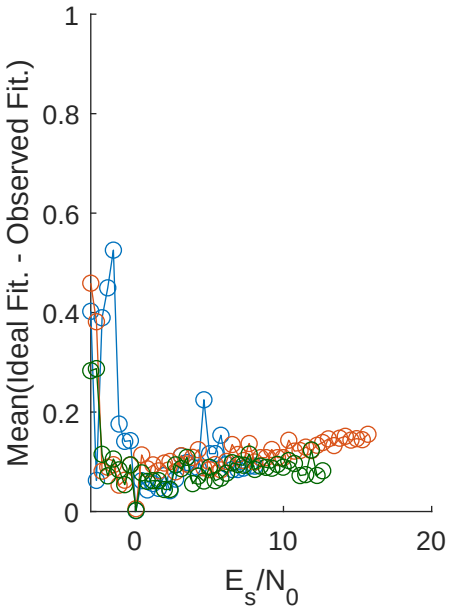


(a) Binned mean for all CE variants during a Great quality pass using Emergency mission.

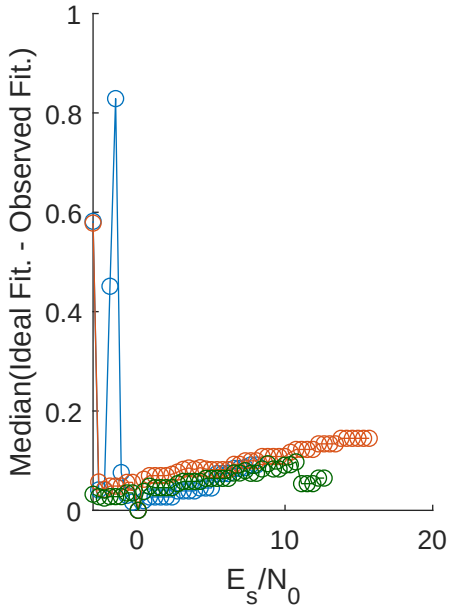


(b) Binned median for all CE variants during a Great quality pass using Emergency mission.

Figure A.41: Binned mean and median plots for all CE variants, using Emergency mission over a Great quality pass.

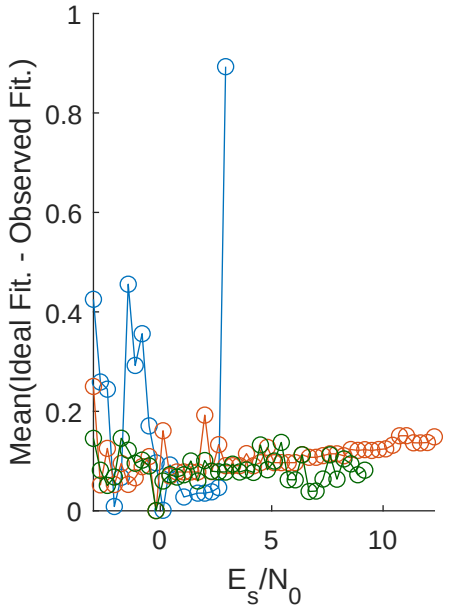


(a) Binned mean for all CE variants during a Good quality pass using Emergency mission.

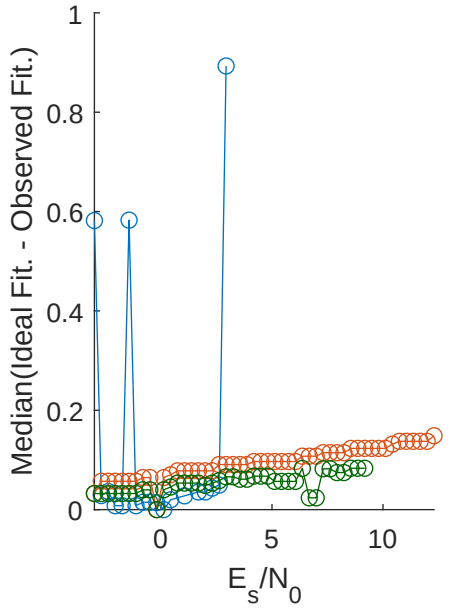


(b) Binned median for all CE variants during a Good quality pass using Emergency mission.

Figure A.42: Binned mean and median plots for all CE variants, using Emergency mission over a Good quality pass.



(a) Binned mean for all CE variants during a Poor quality pass using Emergency mission.



(b) Binned median for all CE variants during a Poor quality pass using Emergency mission.

Figure A.43: Binned mean and median plots for all CE variants, using Emergency mission over a Poor quality pass.

Appendix B

C++ Simulation Selected Results

B.1 C++ Simulation Time Series

B.1.1 Cooperation Mission

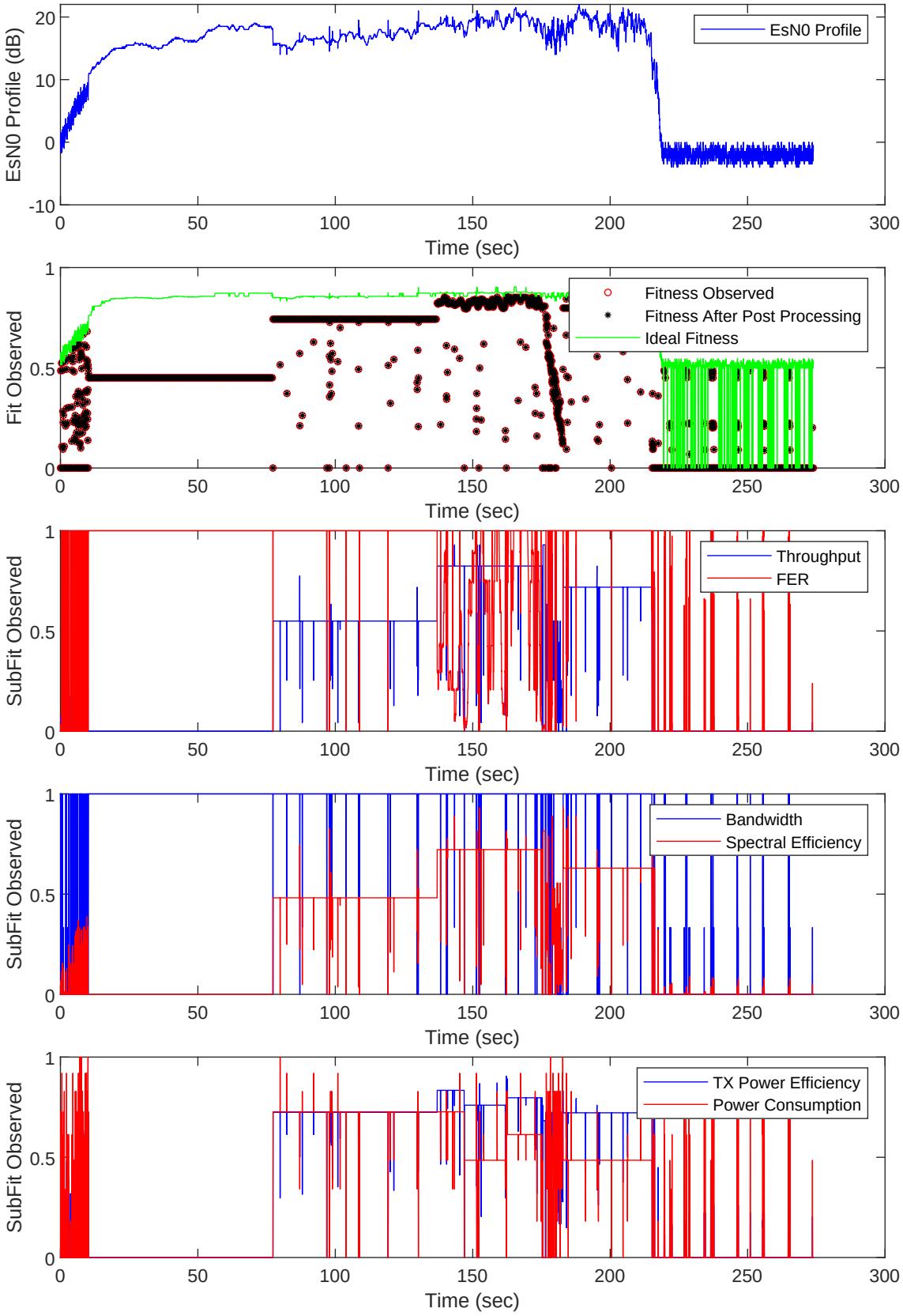


Figure B.1: Operation of CE-LM on SNR profile 1, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

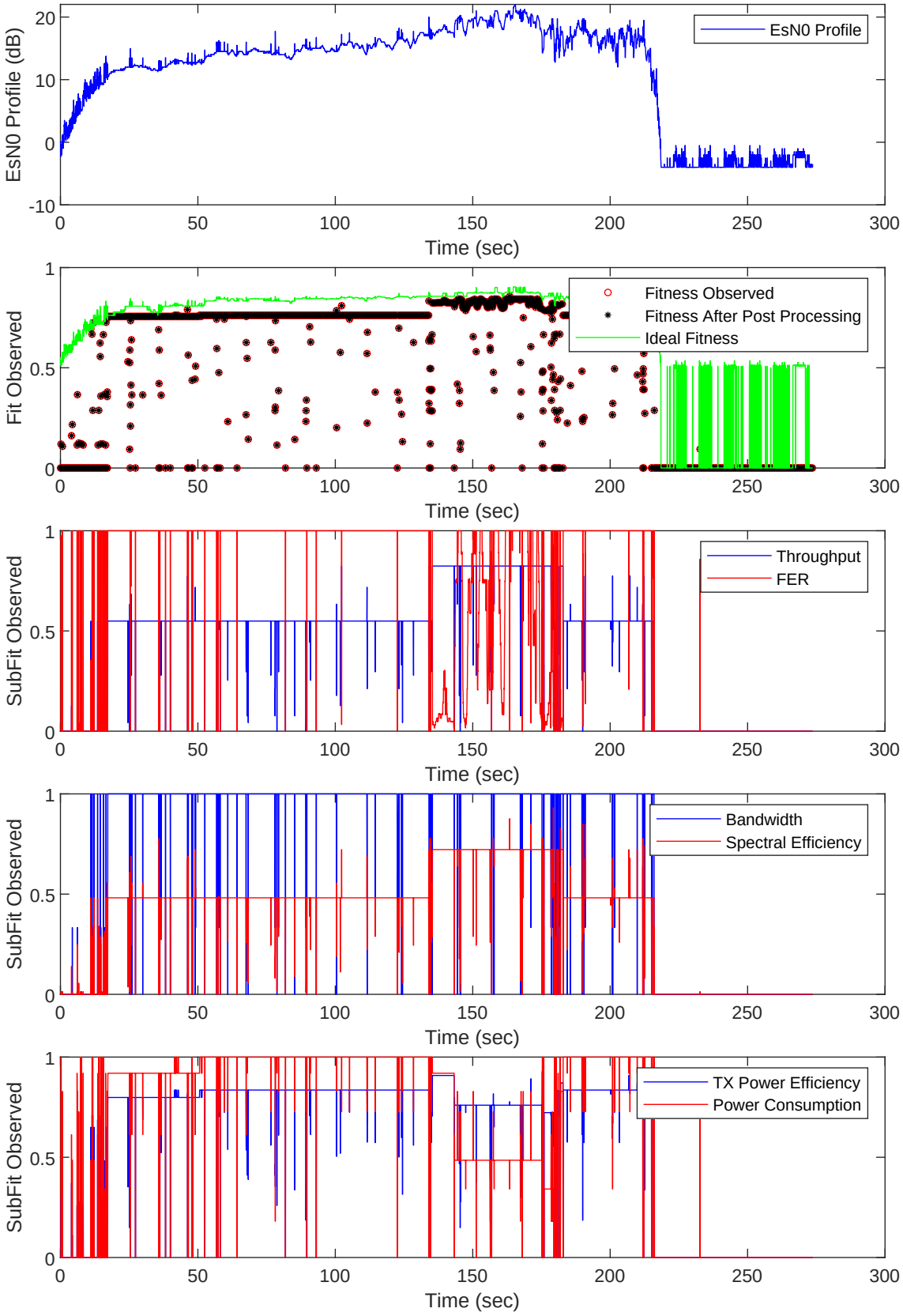


Figure B.2: Operation of CE-RLM on SNR profile 1, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

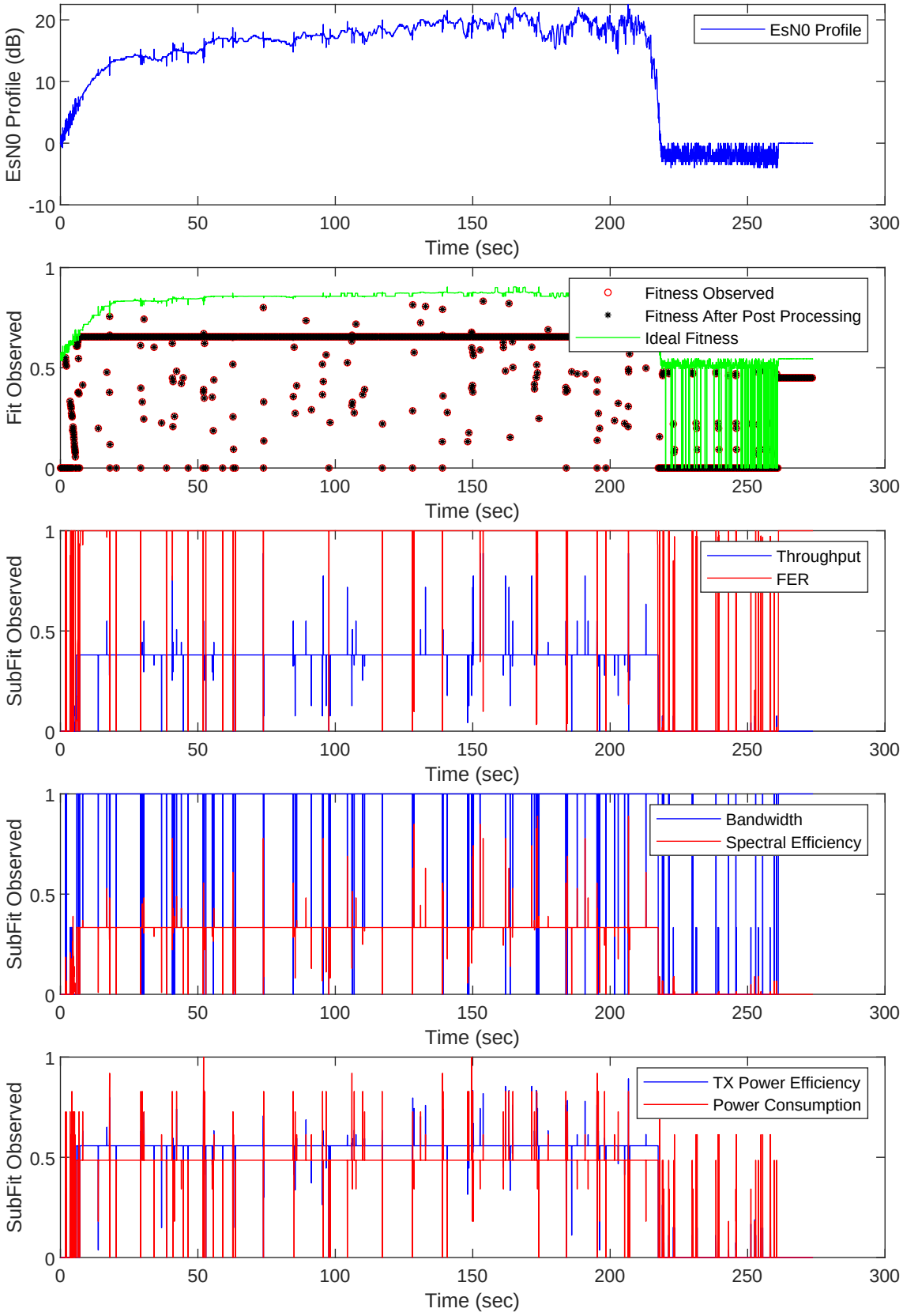


Figure B.3: Operation of CE-NSE on SNR profile 1, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

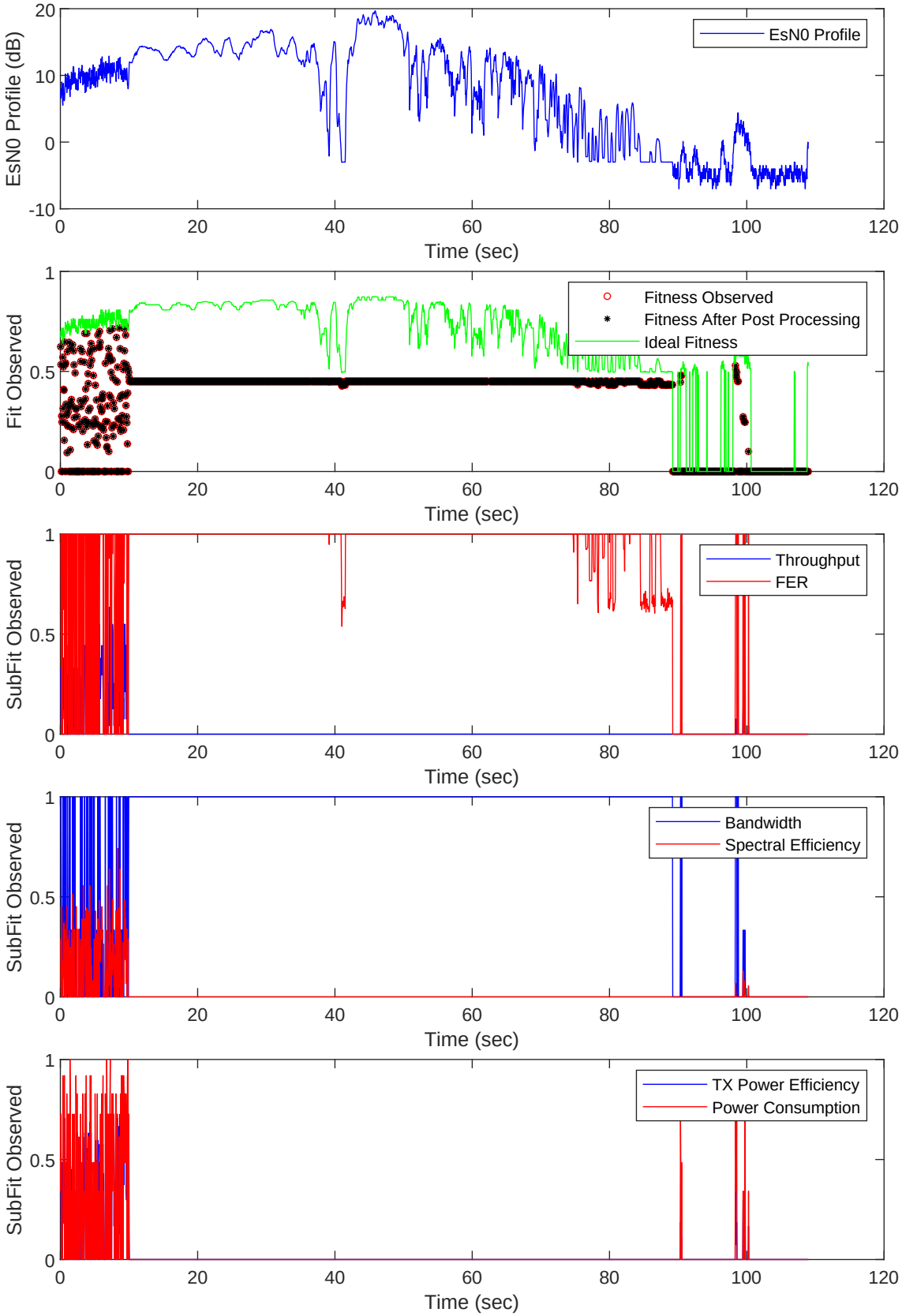


Figure B.4: Operation of CE-LM on SNR profile 2, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

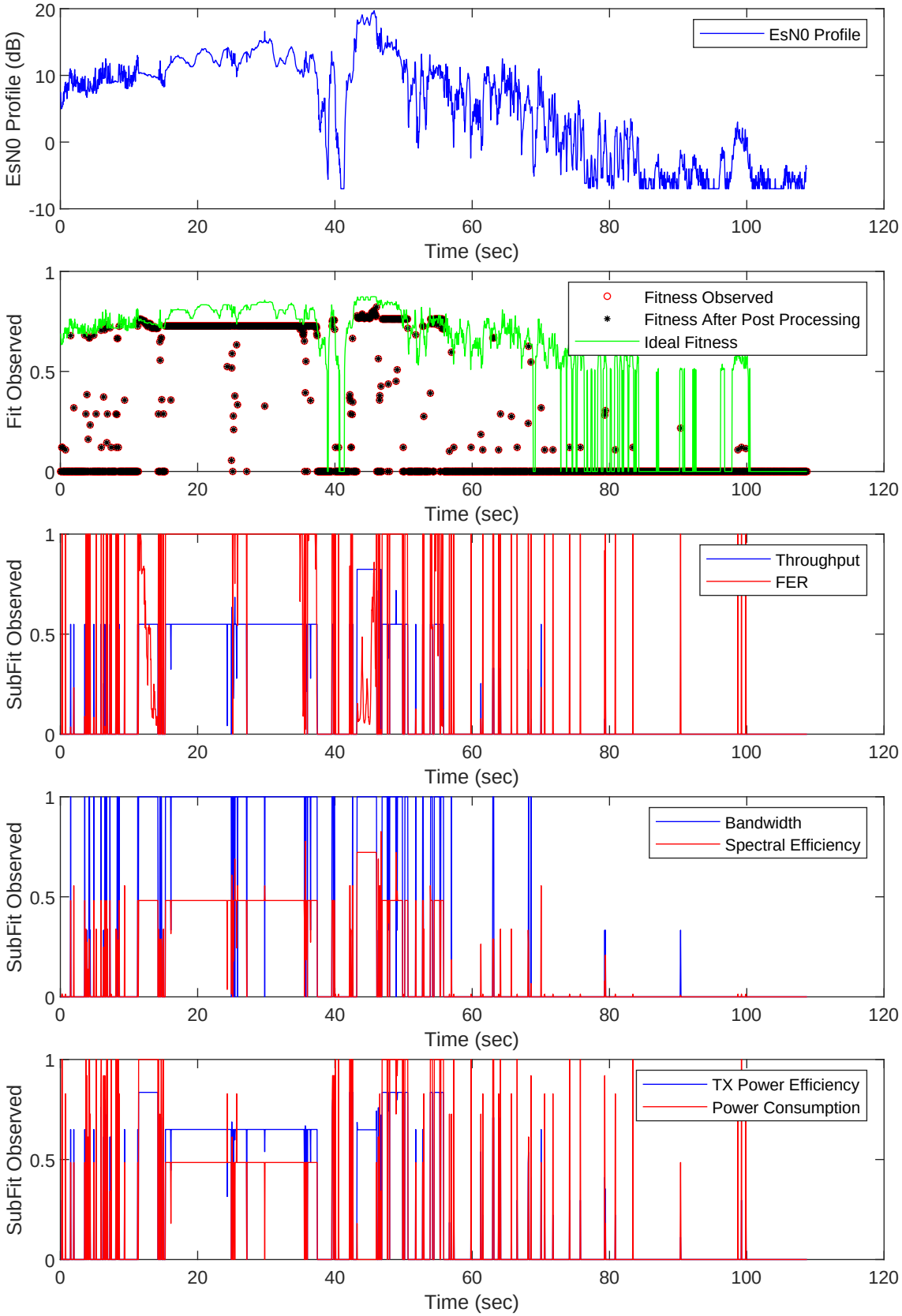


Figure B.5: Operation of CE-RLM on SNR profile 2, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

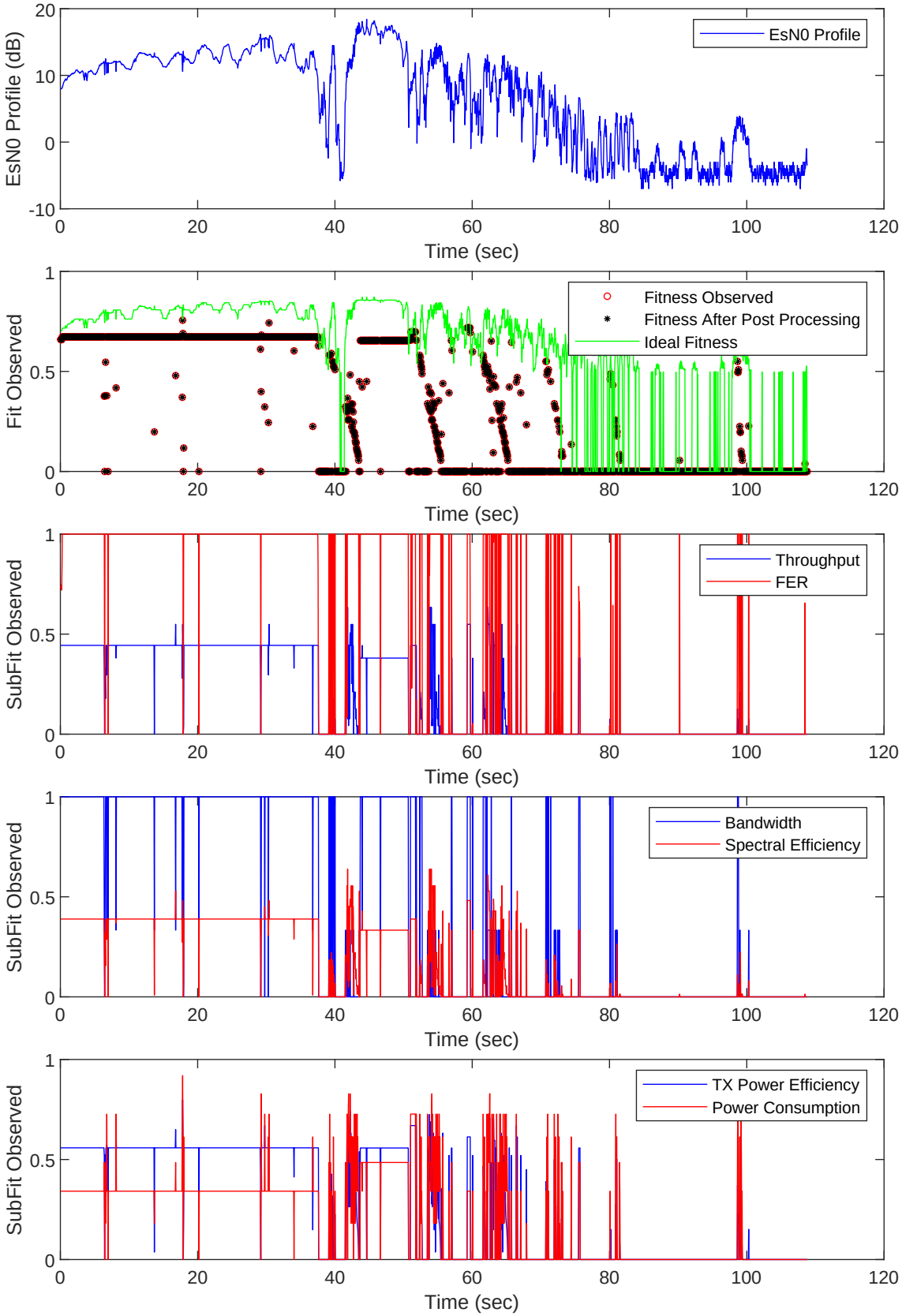


Figure B.6: Operation of CE-NSE on SNR profile 2, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

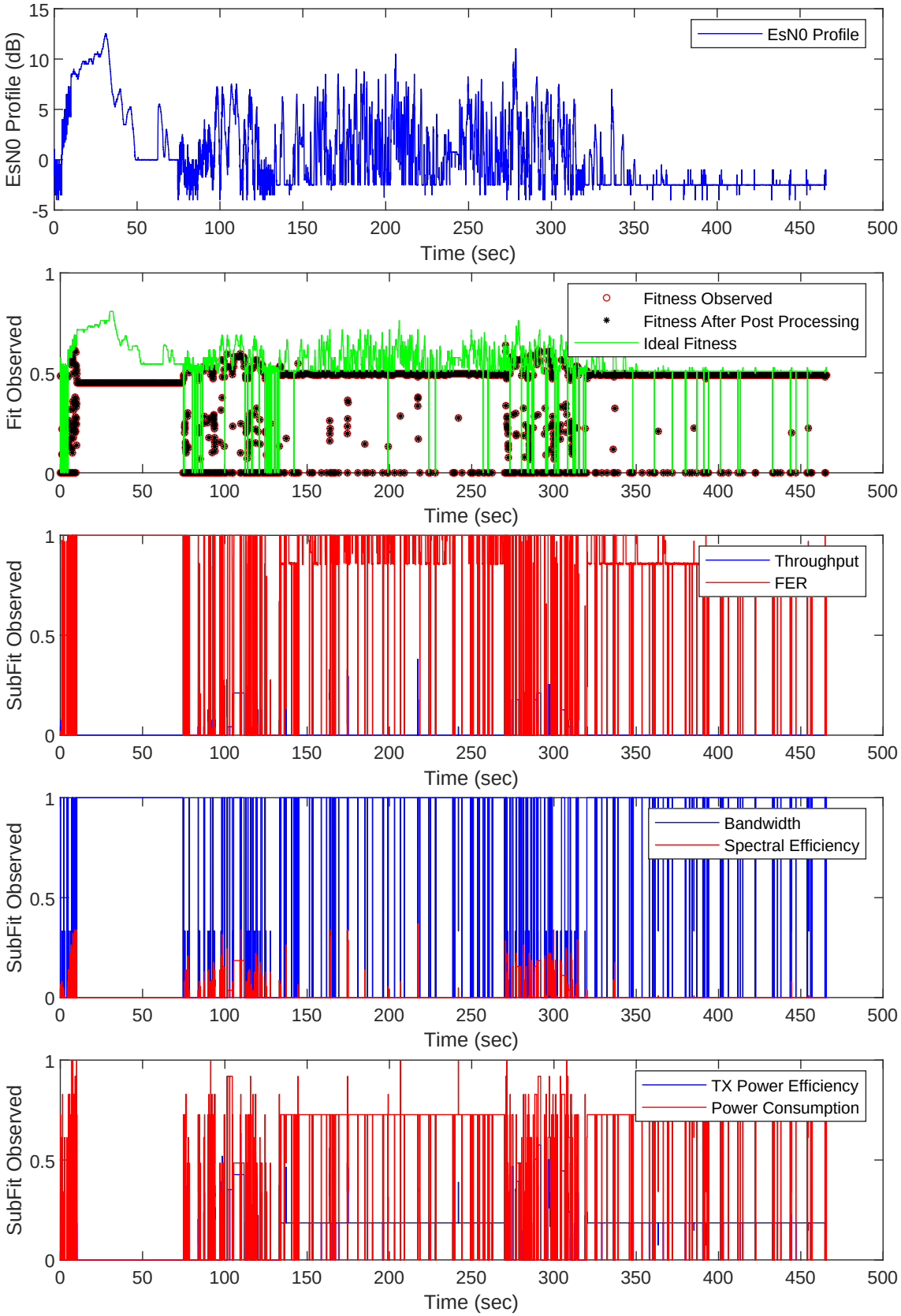


Figure B.7: Operation of CE-LM on SNR profile 3, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

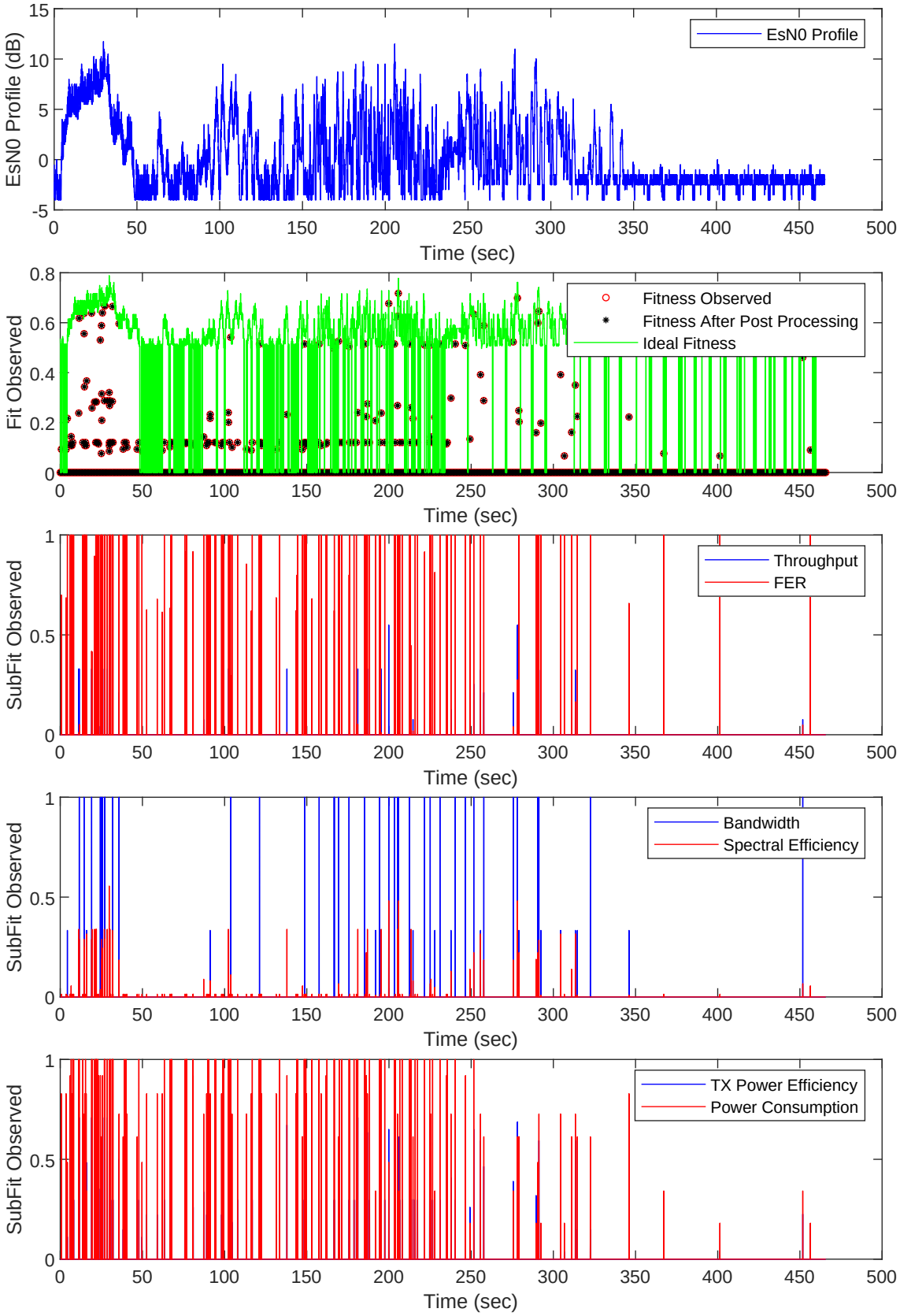


Figure B.8: Operation of CE-RLM on SNR profile 3, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

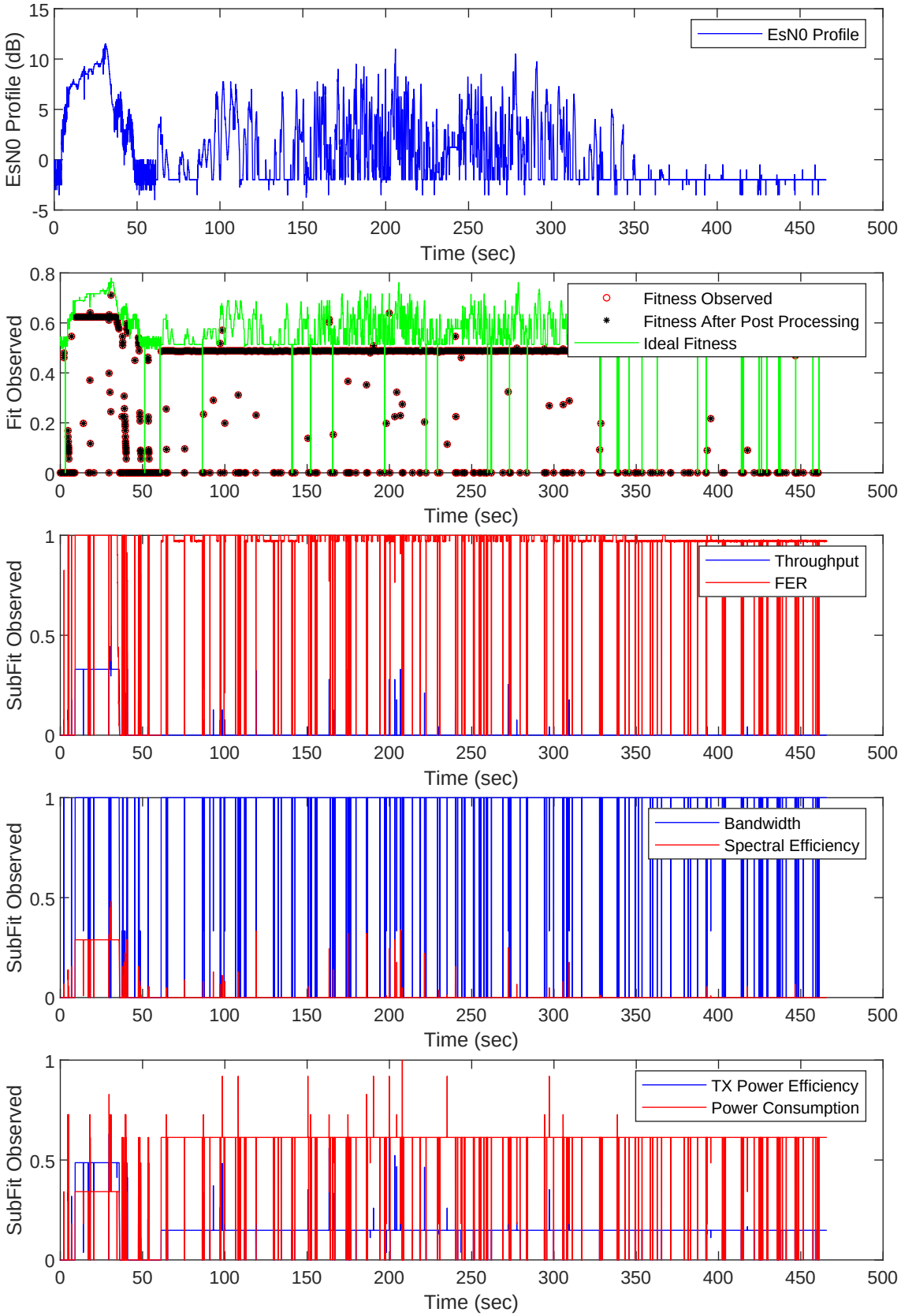


Figure B.9: Operation of CE-NSE on SNR profile 3, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

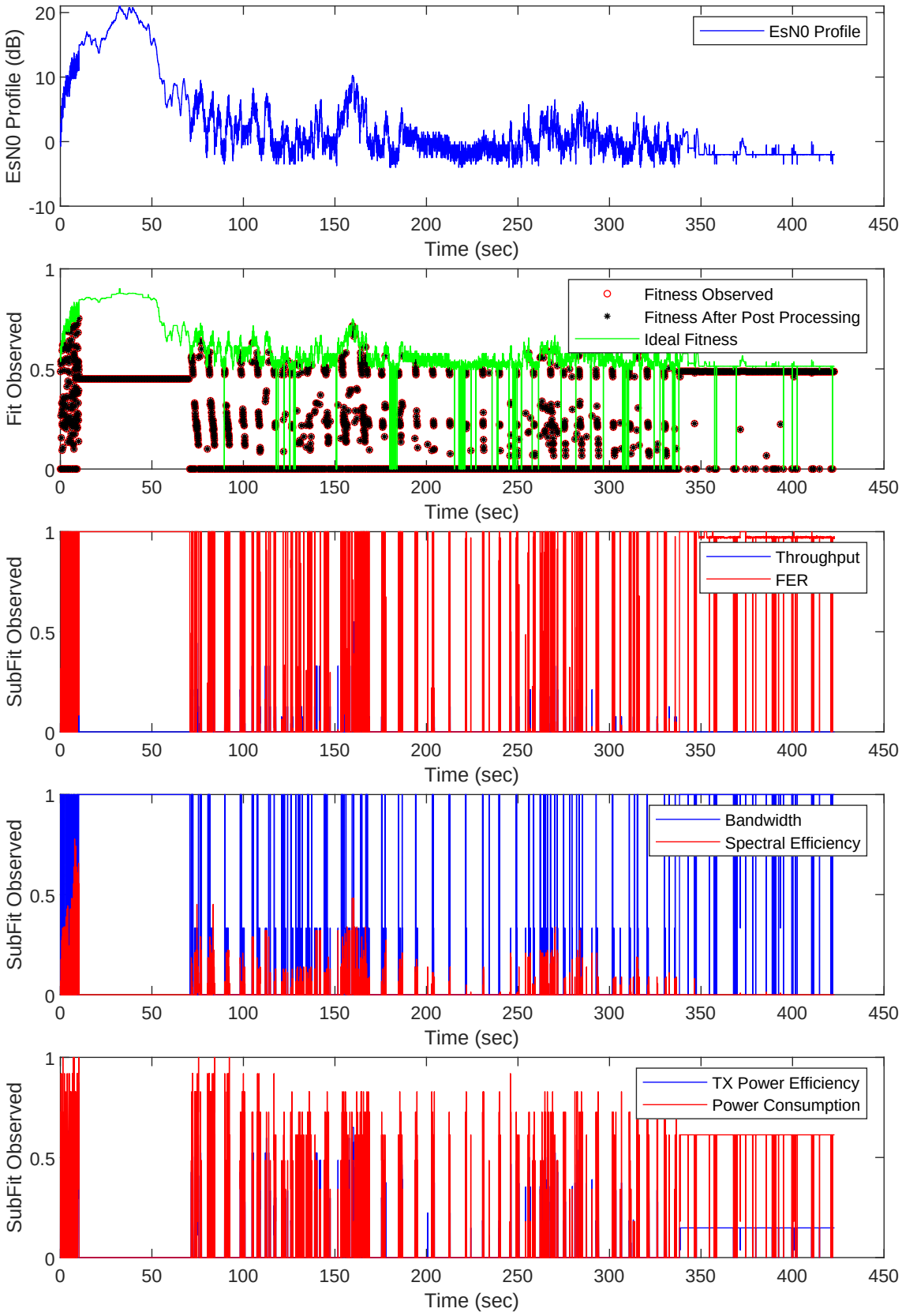


Figure B.10: Operation of CE-LM on SNR profile 4, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

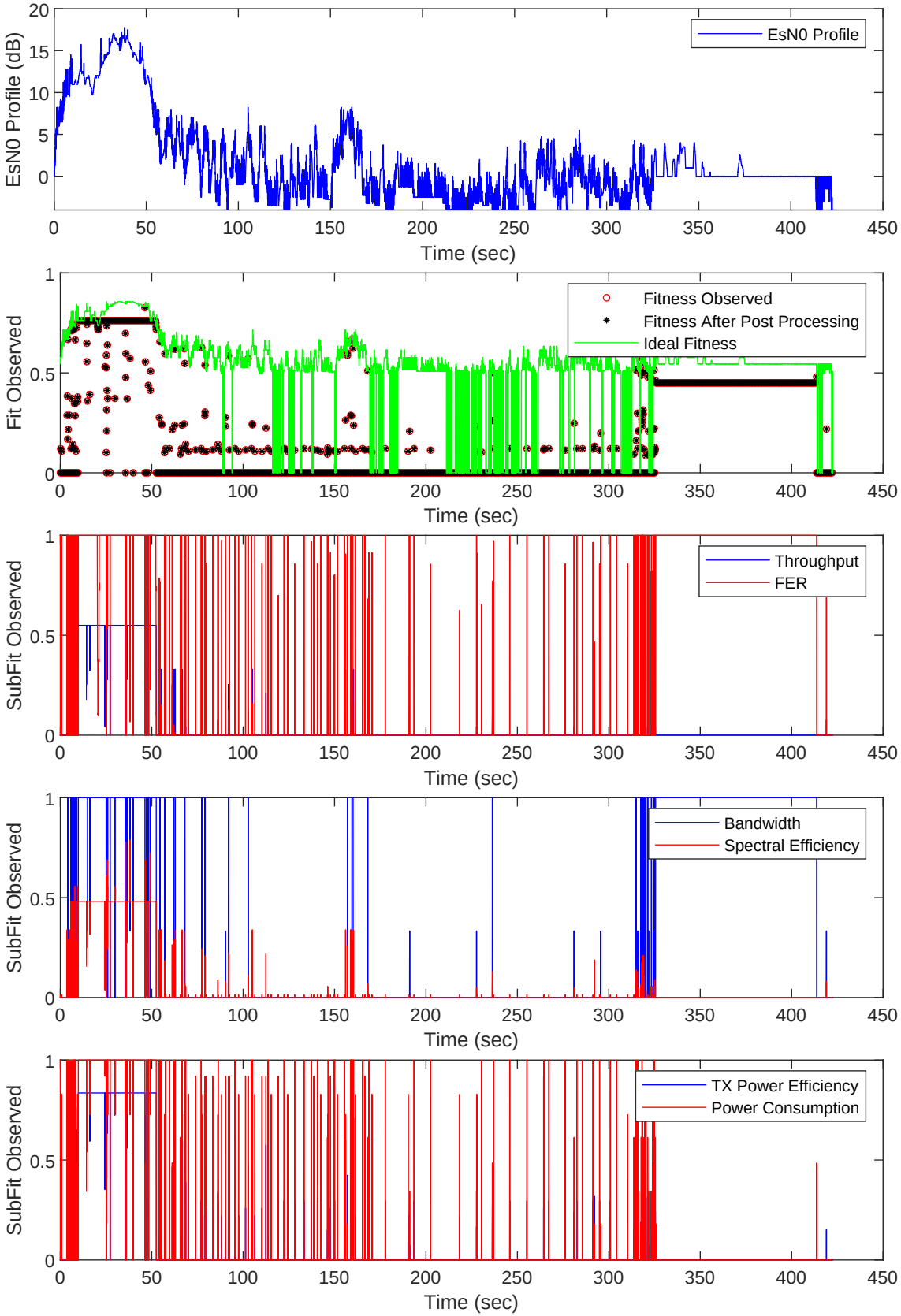


Figure B.11: Operation of CE-RLM on SNR profile 4, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

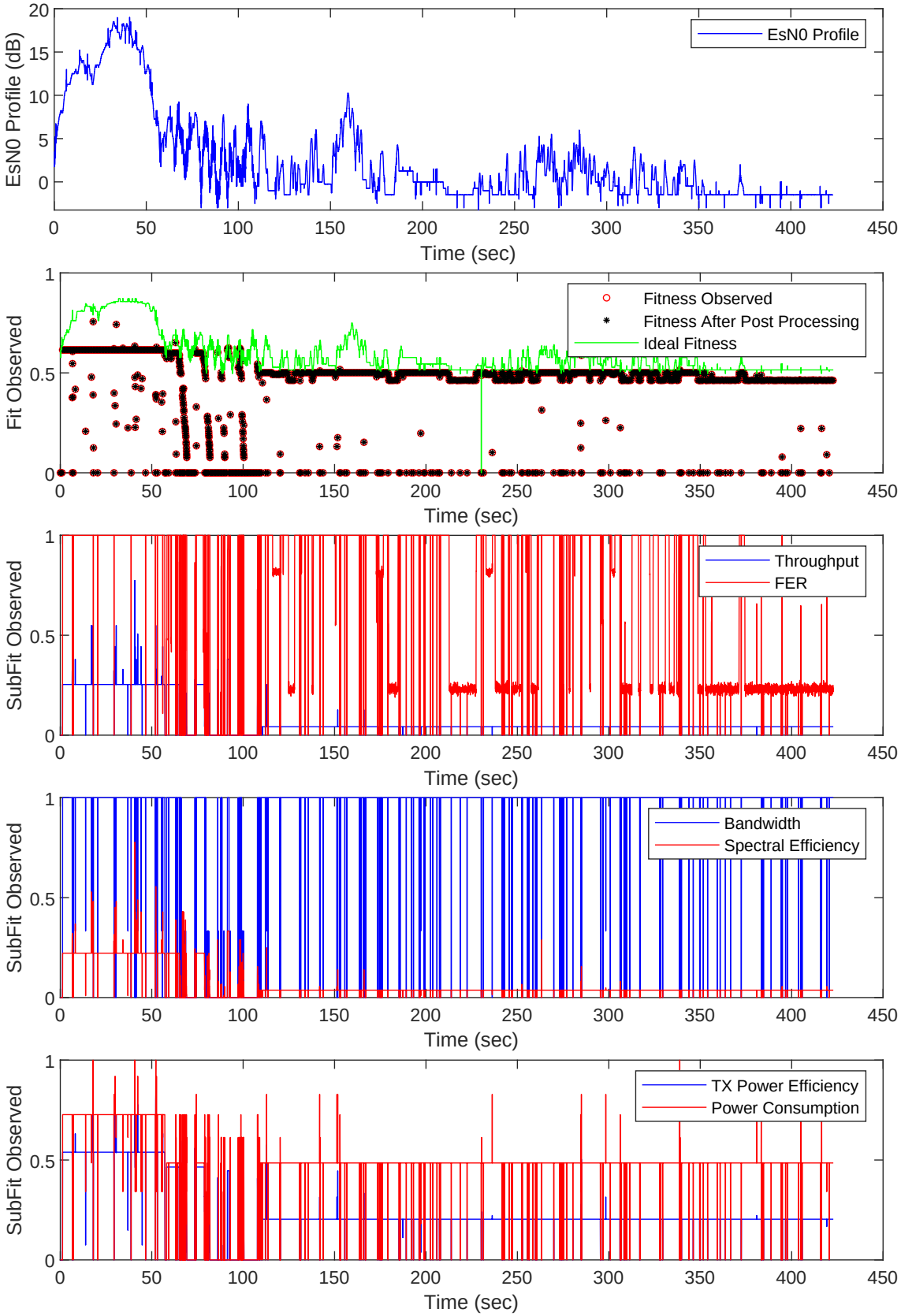


Figure B.12: Operation of CE-NSE on SNR profile 4, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

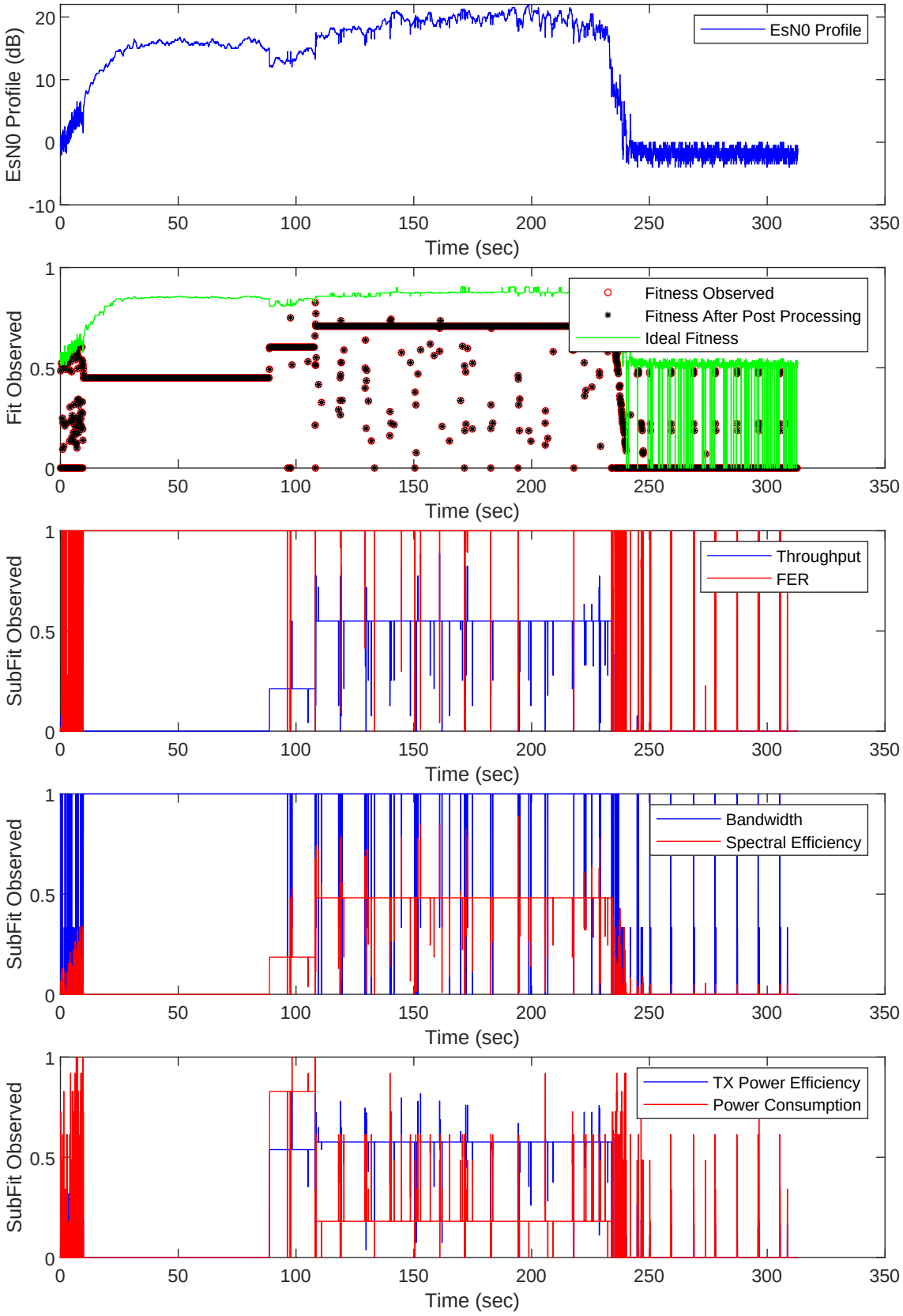


Figure B.13: Operation of CE-LM on SNR profile 5, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

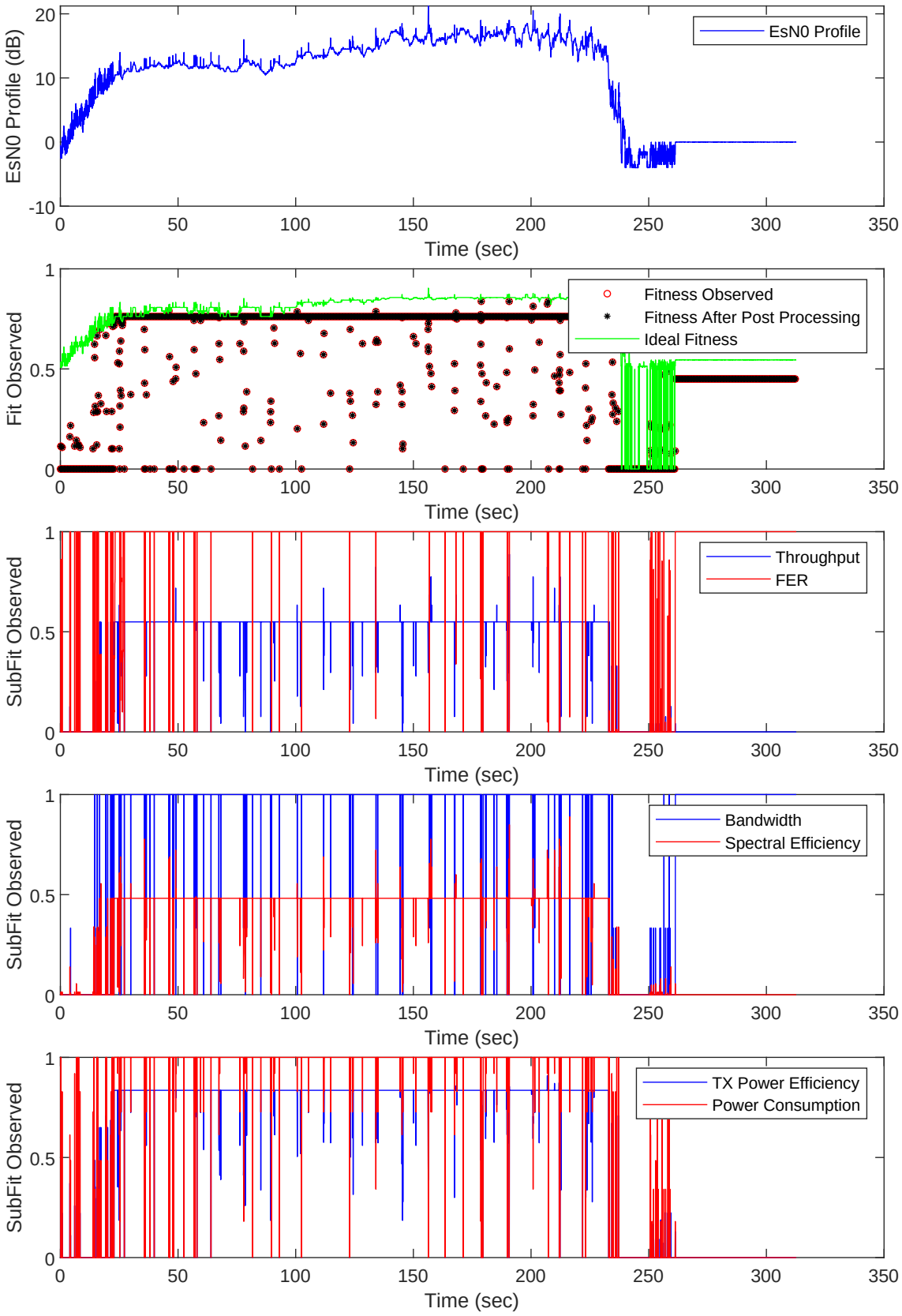


Figure B.14: Operation of CE-RLM on SNR profile 5, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

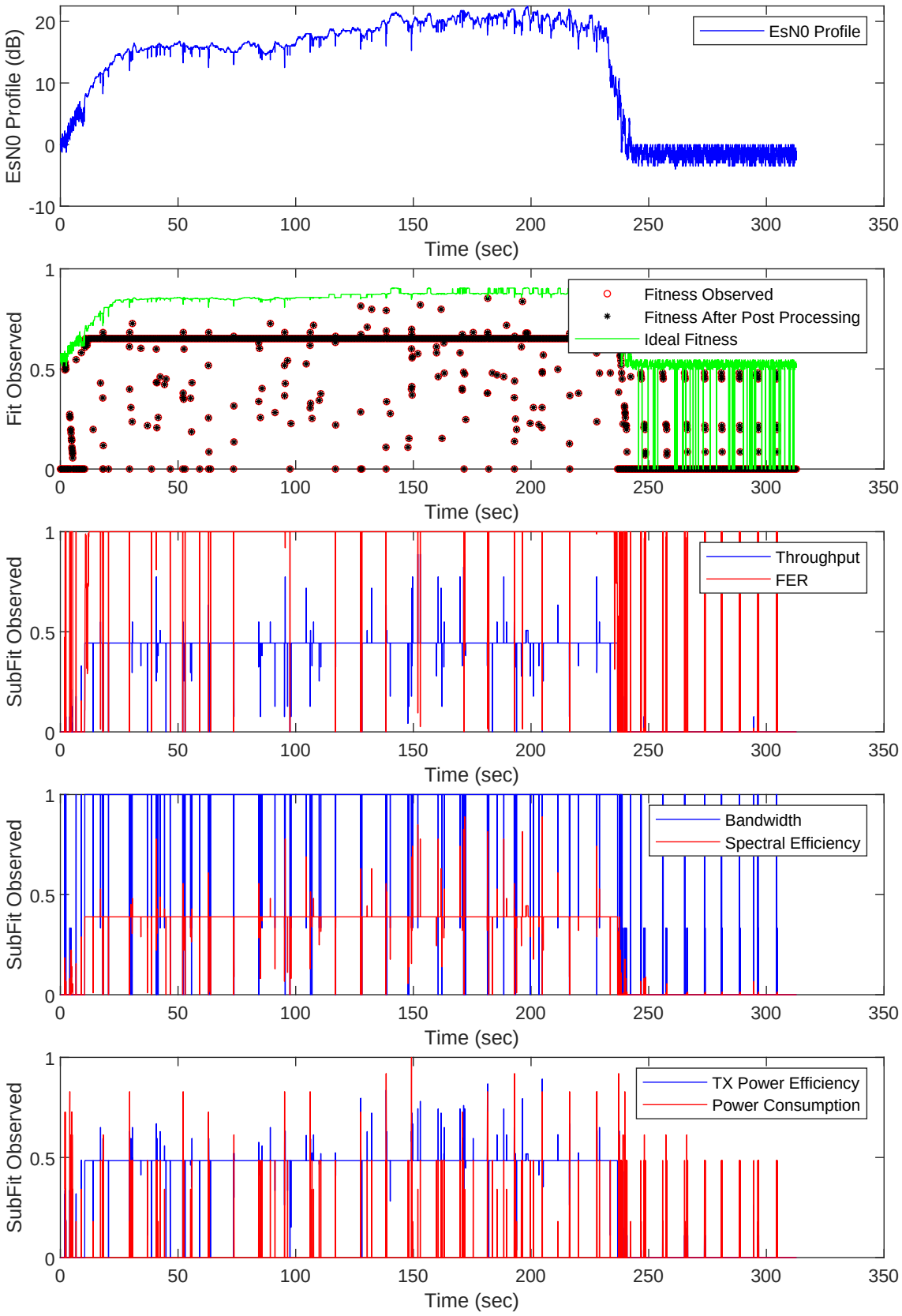


Figure B.15: Operation of CE-NSE on SNR profile 5, using the Cooperation mission. Includes SNR profile, fitness observed, and subfitness values observed.

B.1.2 Power Saving Mission

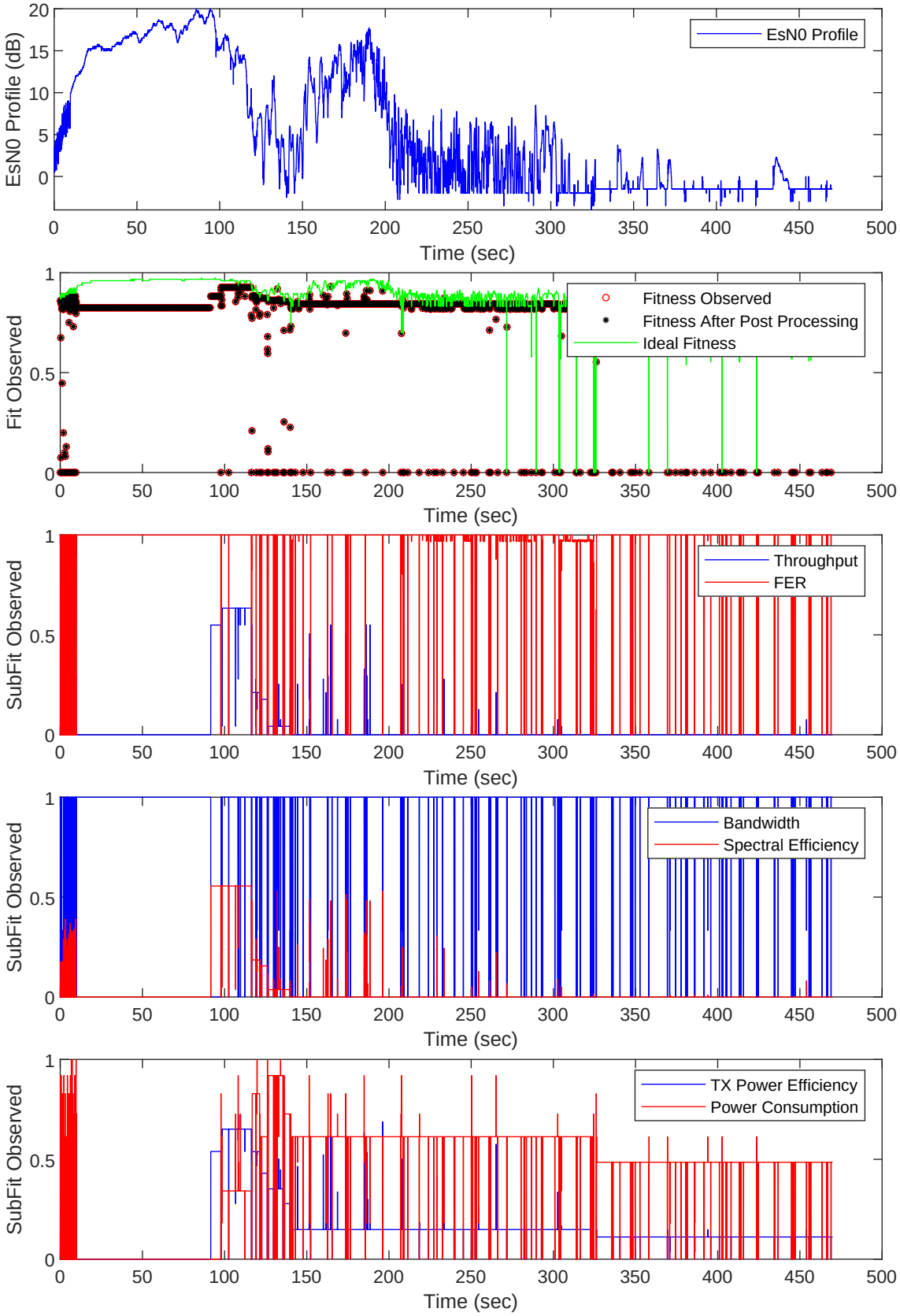


Figure B.16: Operation of CE-LM on SNR profile 1, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

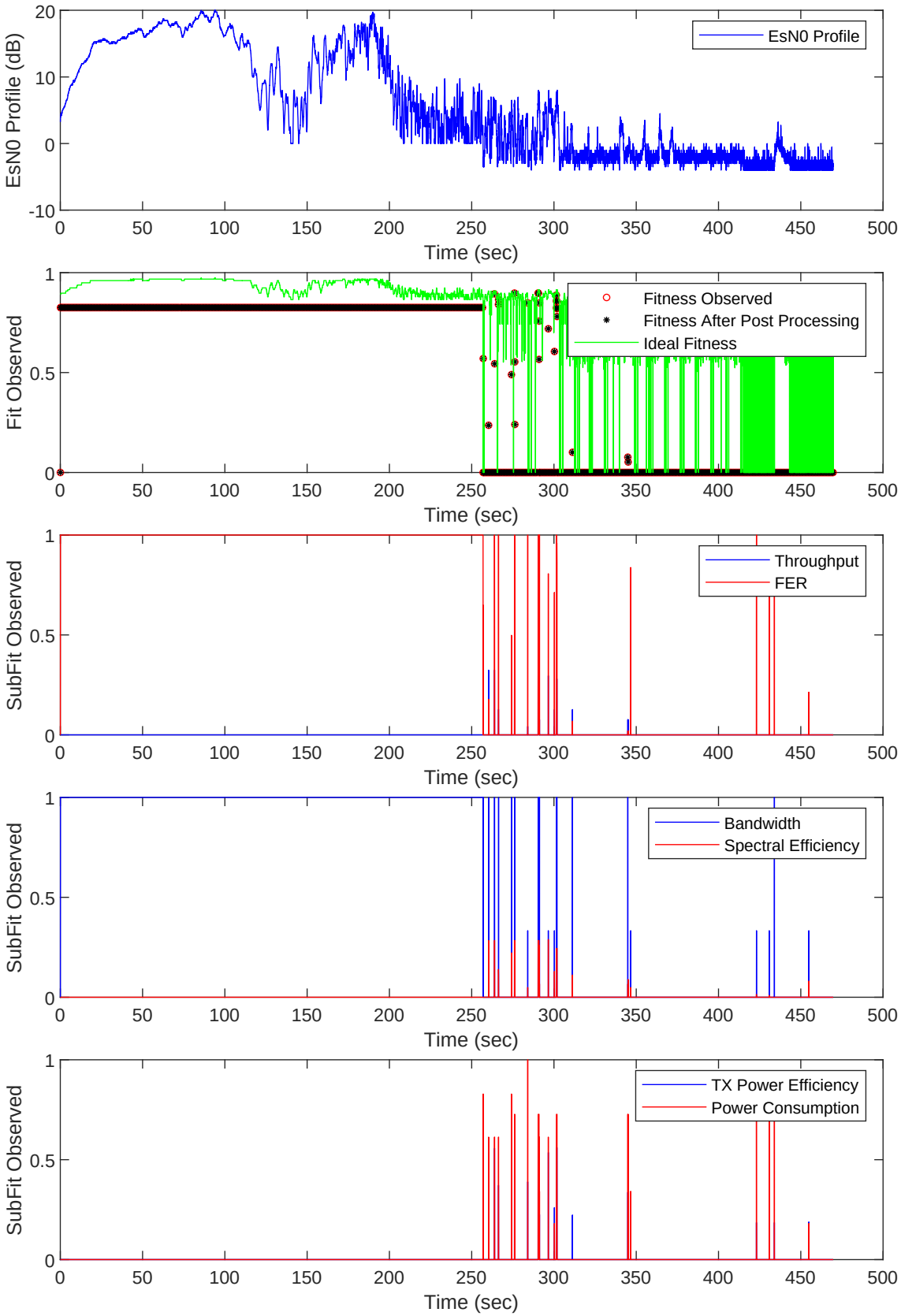


Figure B.17: Operation of CE-RLM on SNR profile 1, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

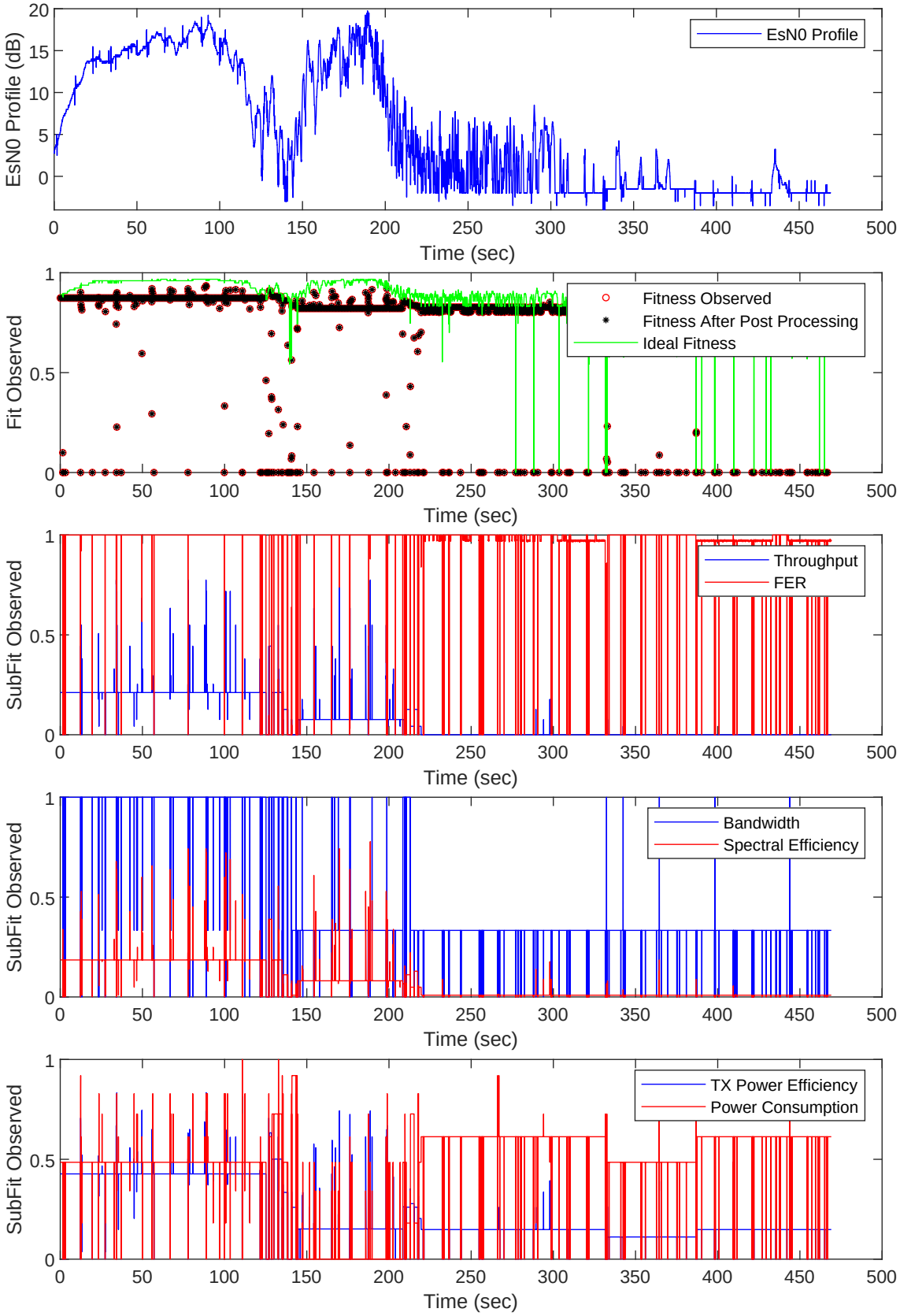


Figure B.18: Operation of CE-NSE on SNR profile 1, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

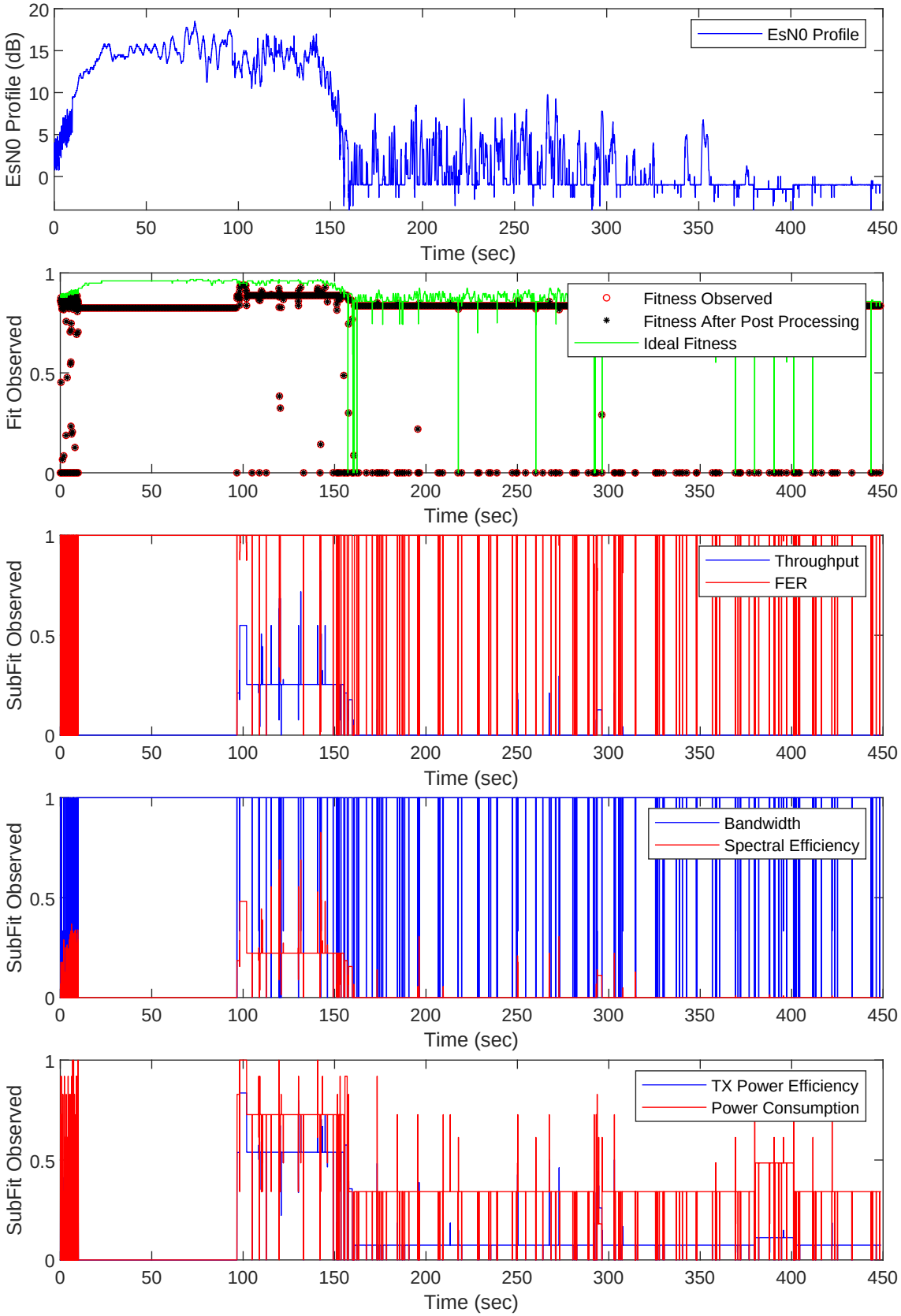


Figure B.19: Operation of CE-LM on SNR profile 2, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

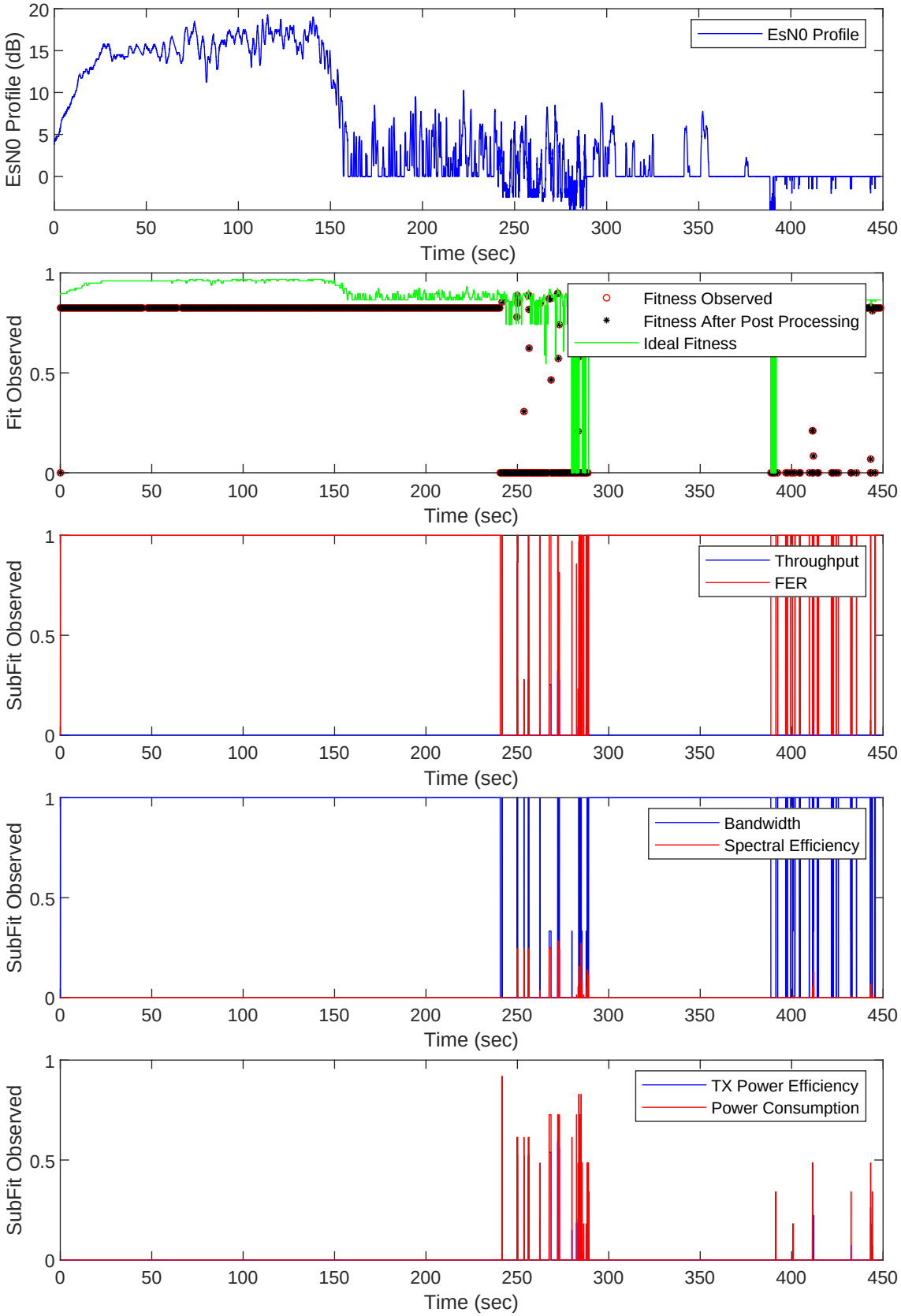


Figure B.20: Operation of CE-RLM on SNR profile 2, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

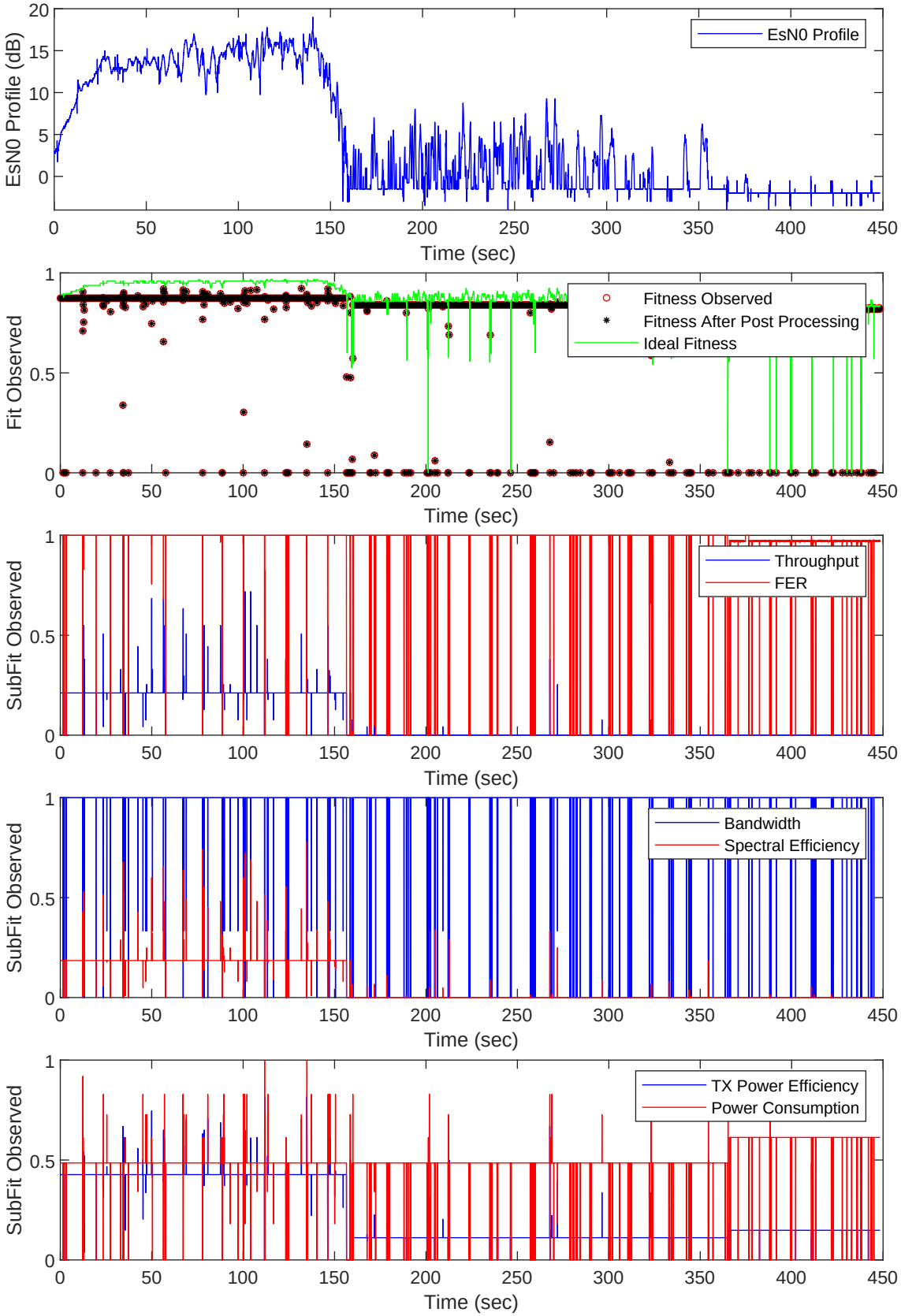


Figure B.21: Operation of CE-NSE on SNR profile 2, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

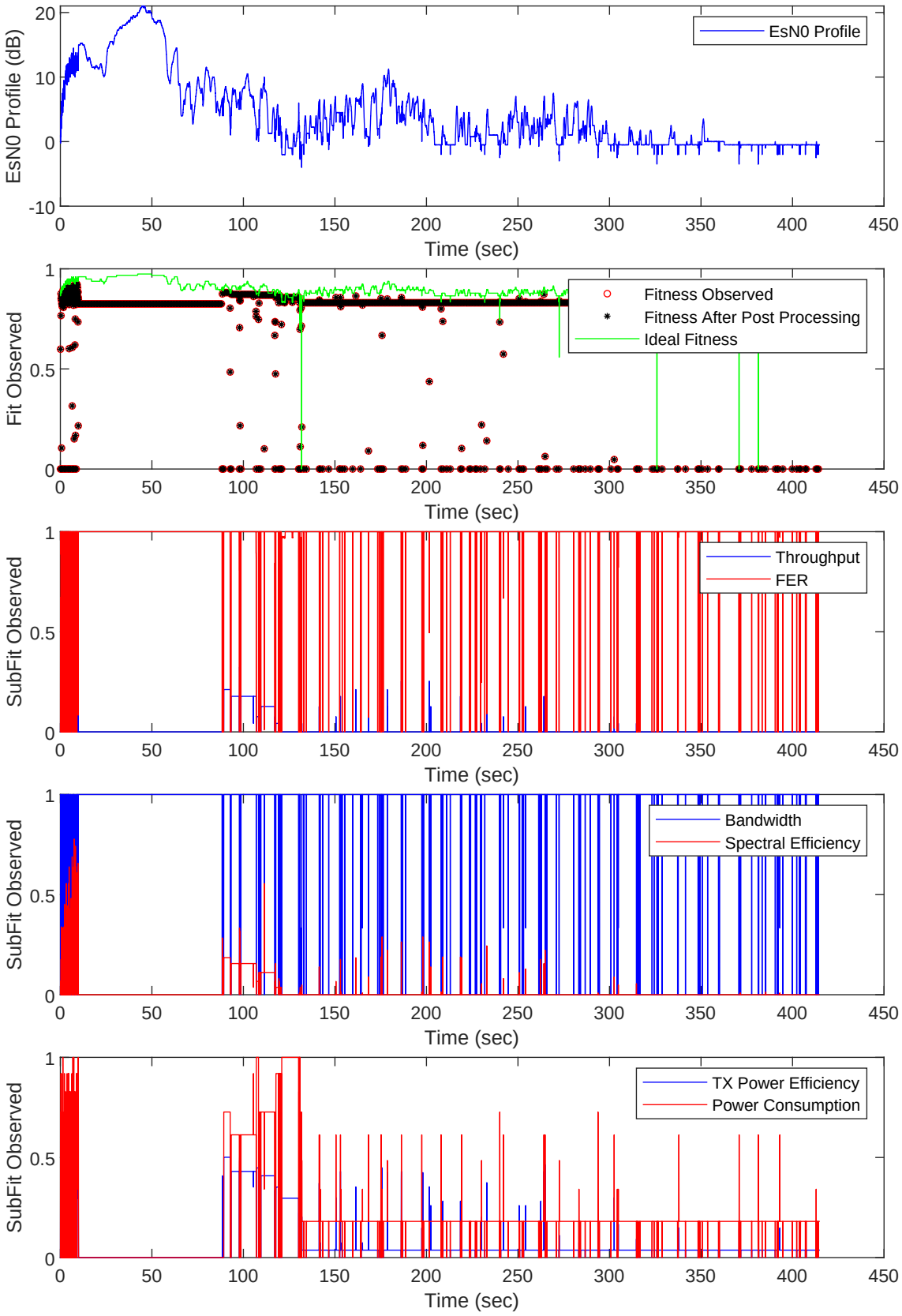


Figure B.22: Operation of CE-LM on SNR profile 3, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

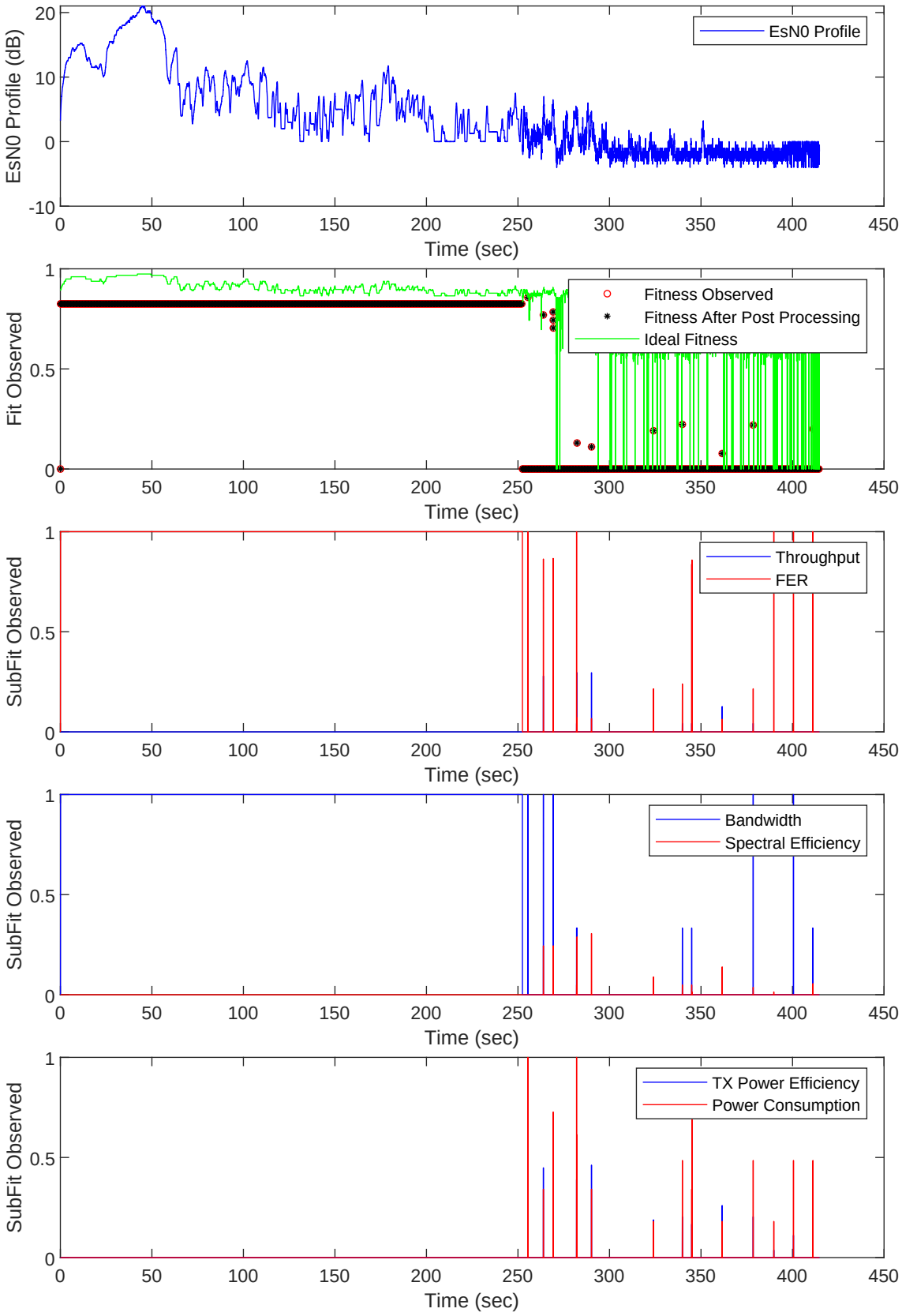


Figure B.23: Operation of CE-RLM on SNR profile 3, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

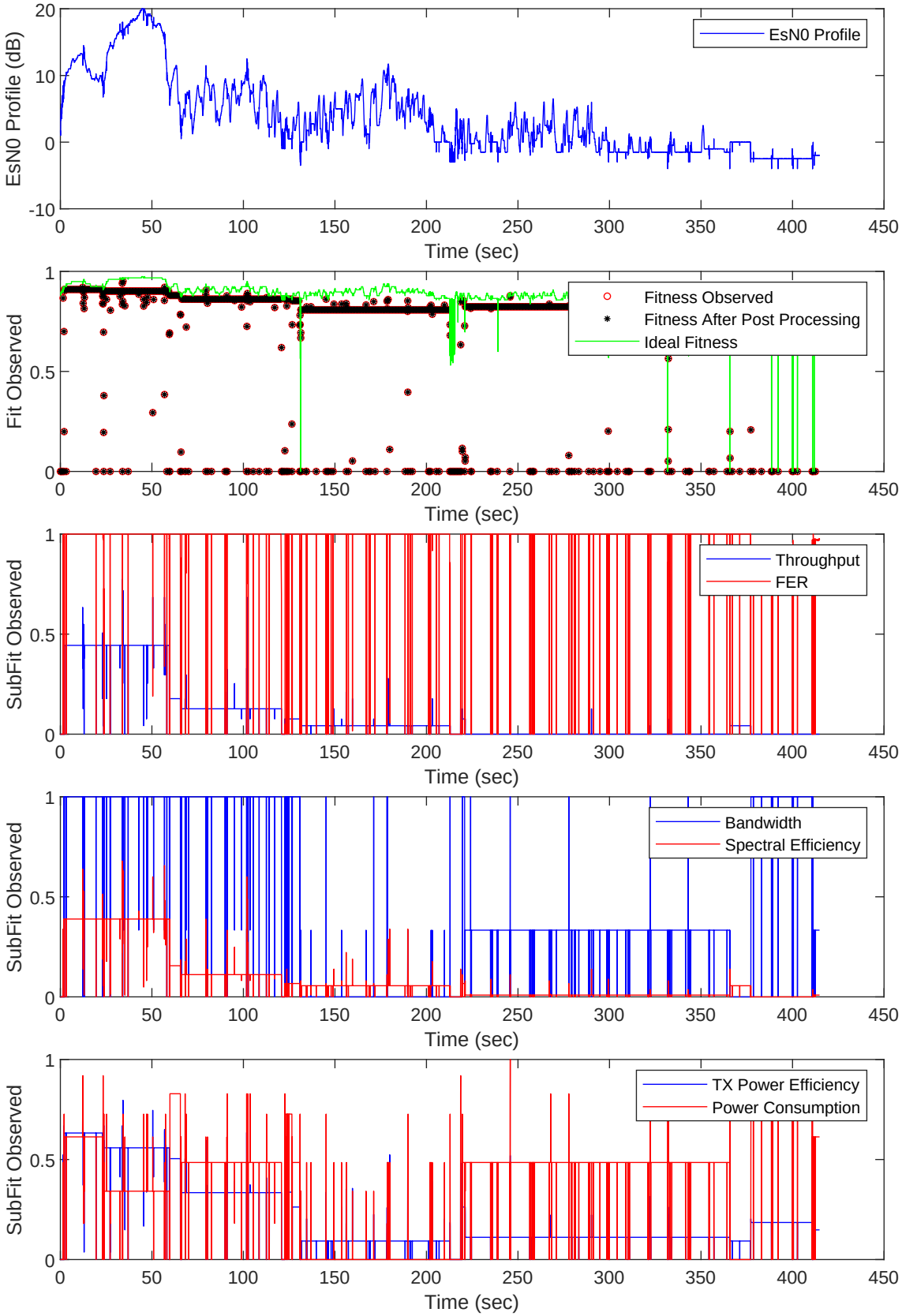


Figure B.24: Operation of CE-NSE on SNR profile 3, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

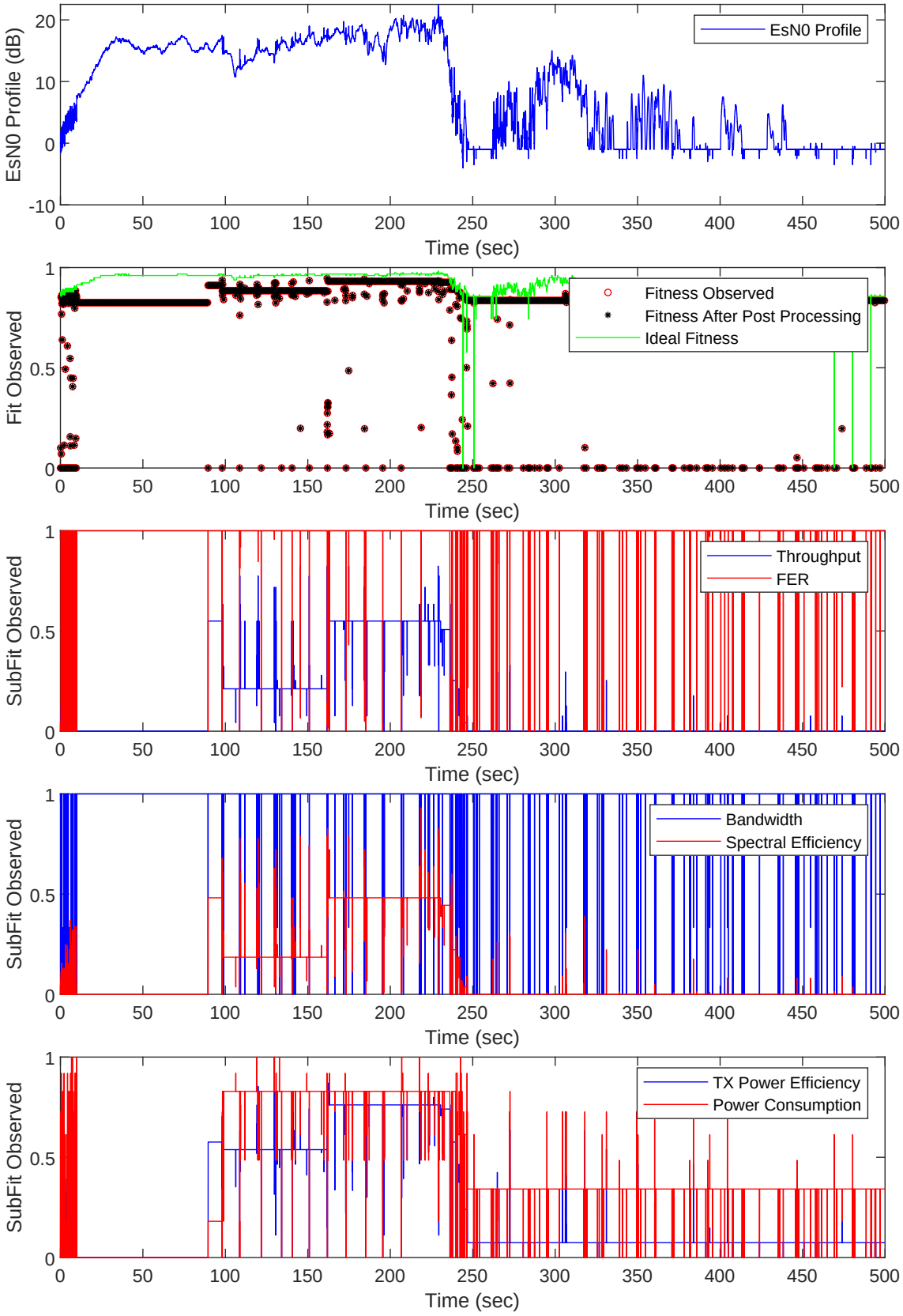


Figure B.25: Operation of CE-LM on SNR profile 4, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

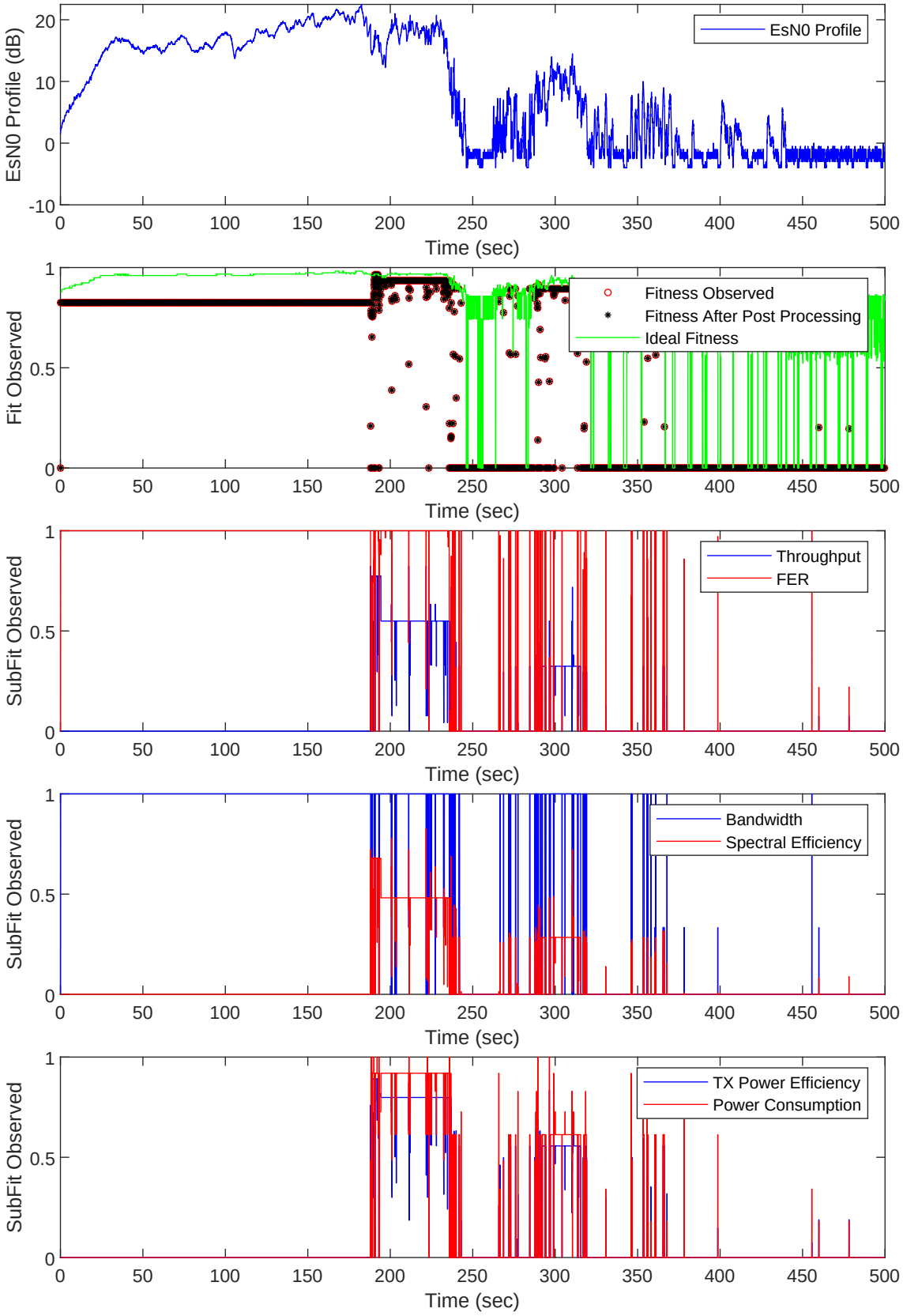


Figure B.26: Operation of CE-RLM on SNR profile 4, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

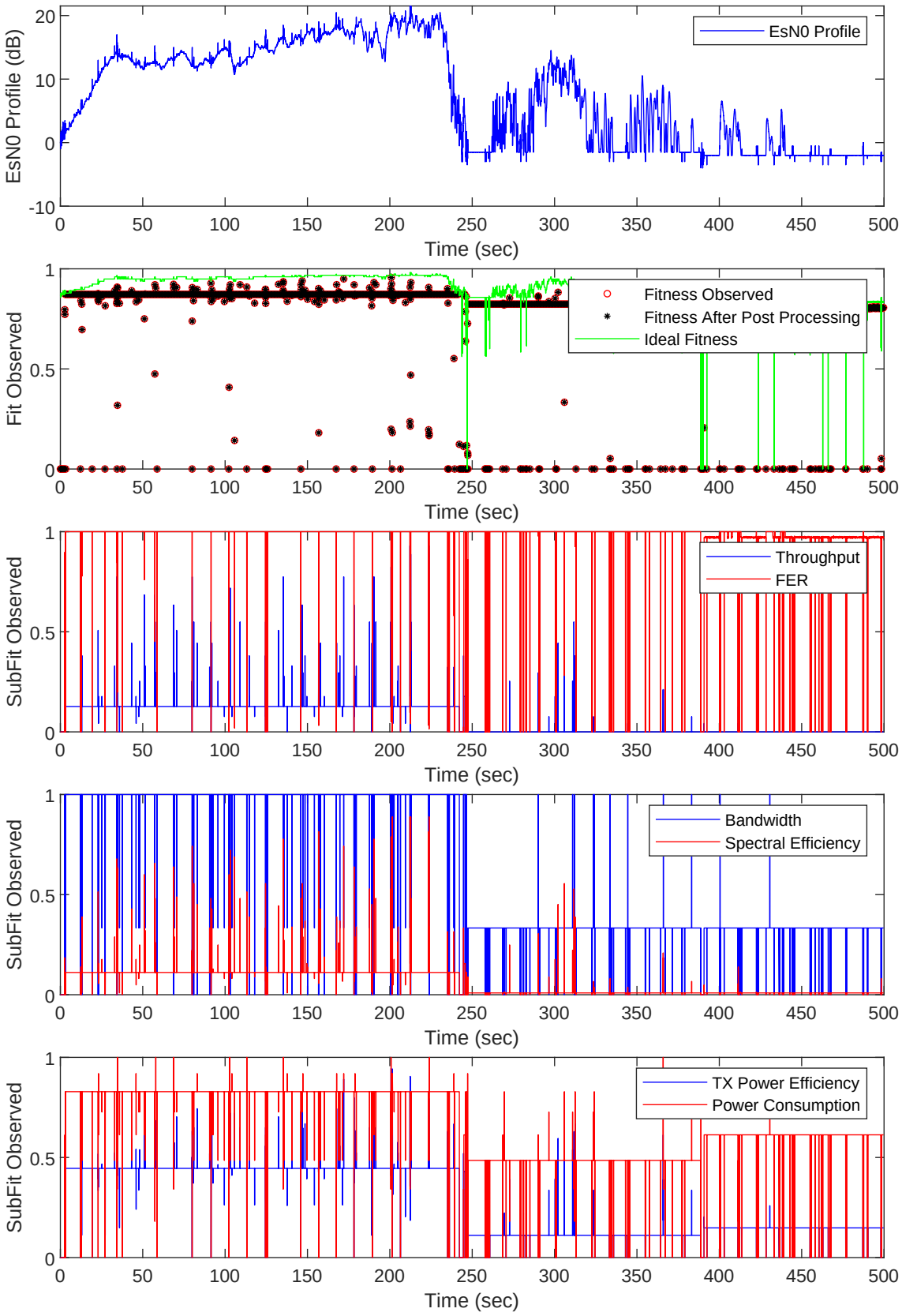


Figure B.27: Operation of CE-NSE on SNR profile 4, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

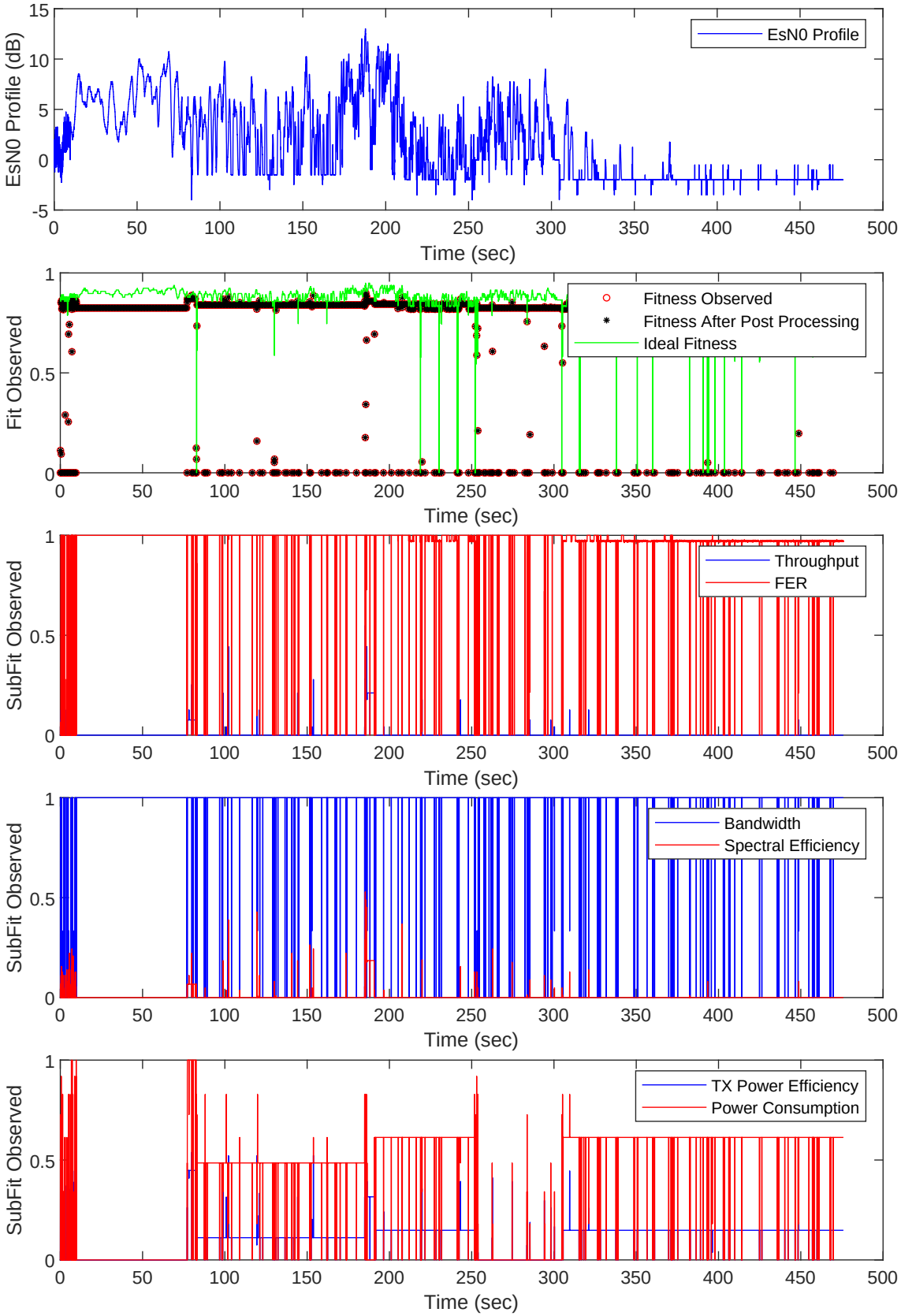


Figure B.28: Operation of CE-LM on SNR profile 5, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

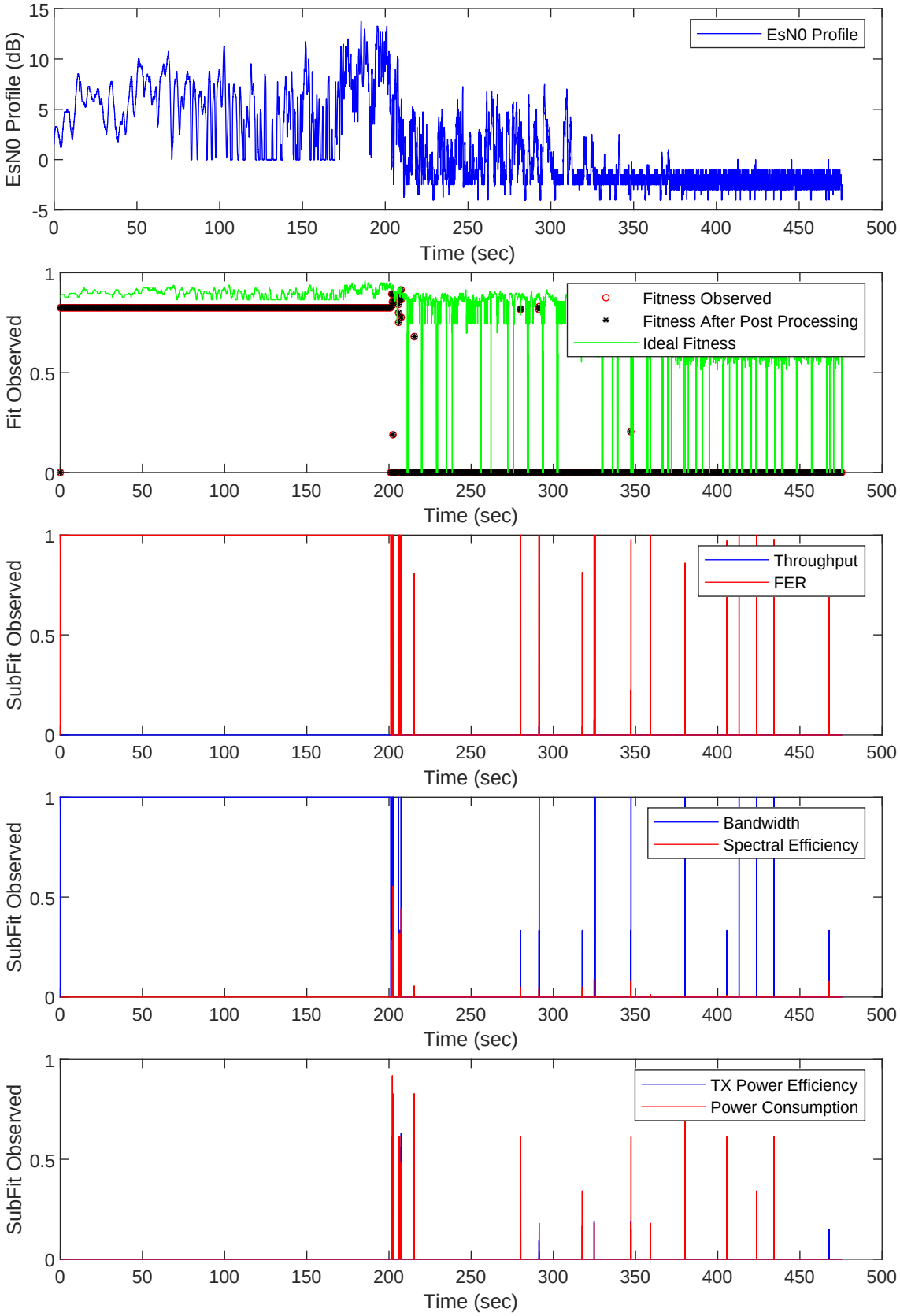


Figure B.29: Operation of CE-RLM on SNR profile 5, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

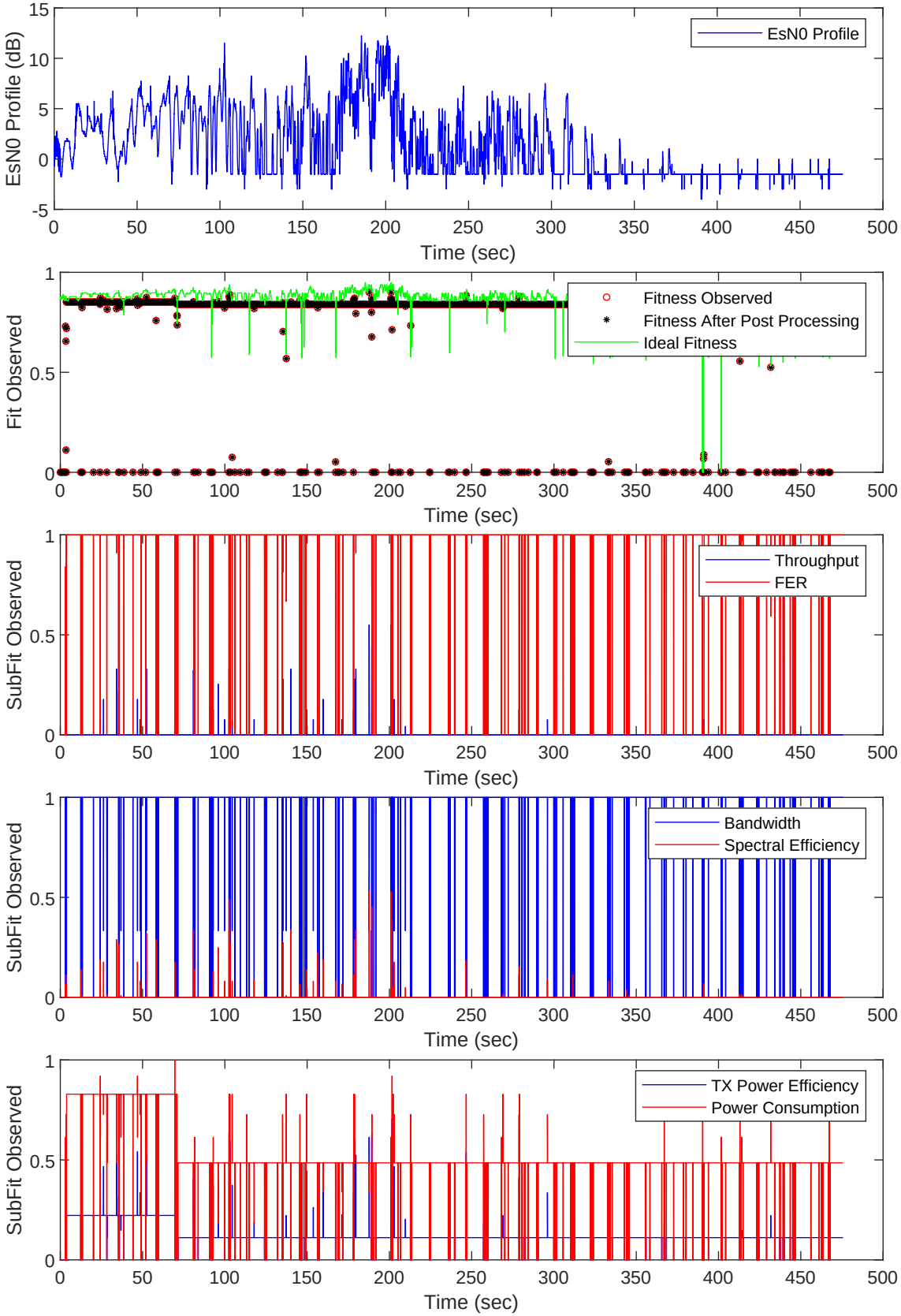


Figure B.30: Operation of CE-NSE on SNR profile 5, using the Power Saving mission. Includes SNR profile, fitness observed, and subfitness values observed.

B.1.3 Emergency Mission

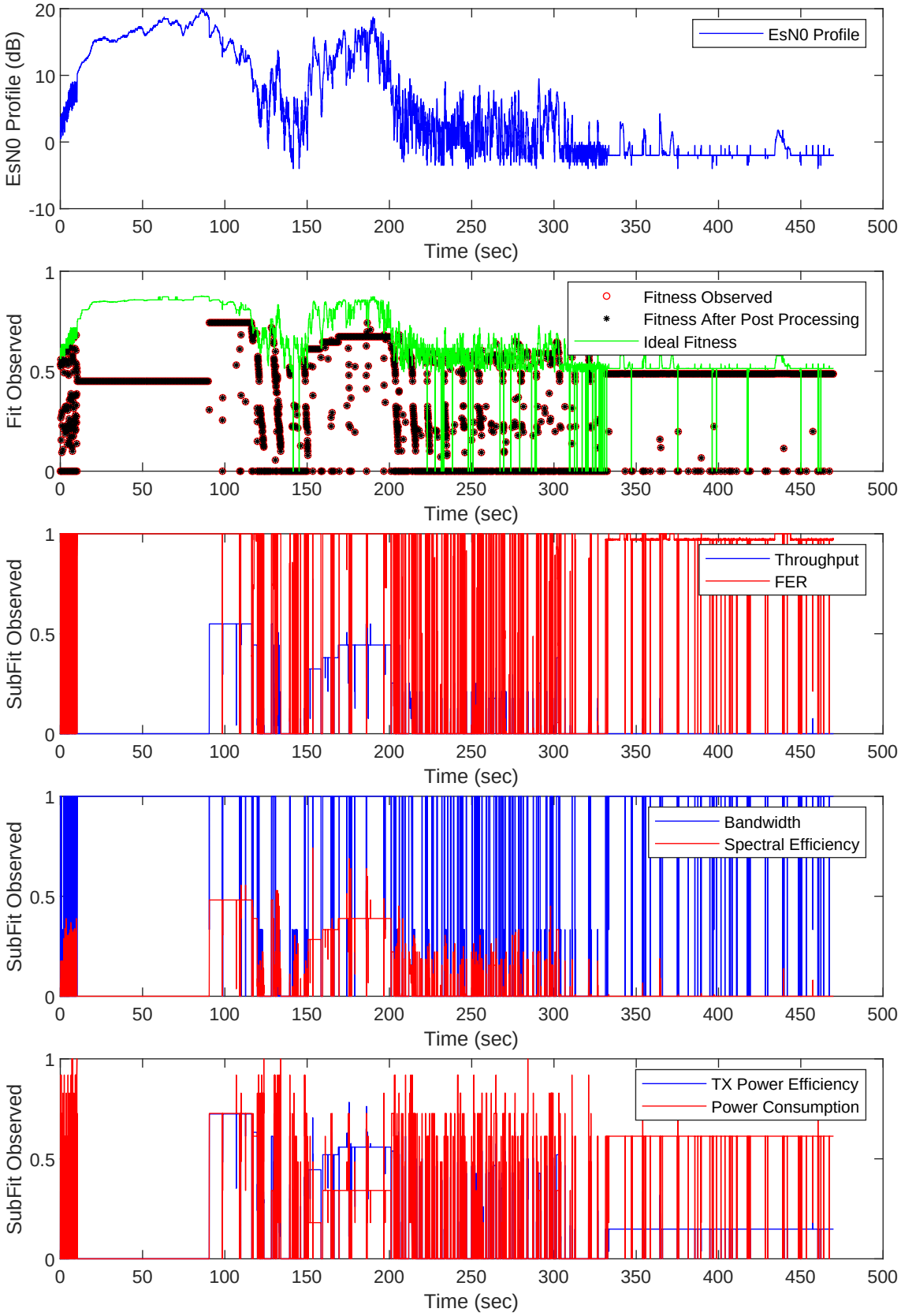


Figure B.31: Operation of CE-LM on SNR profile 1, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

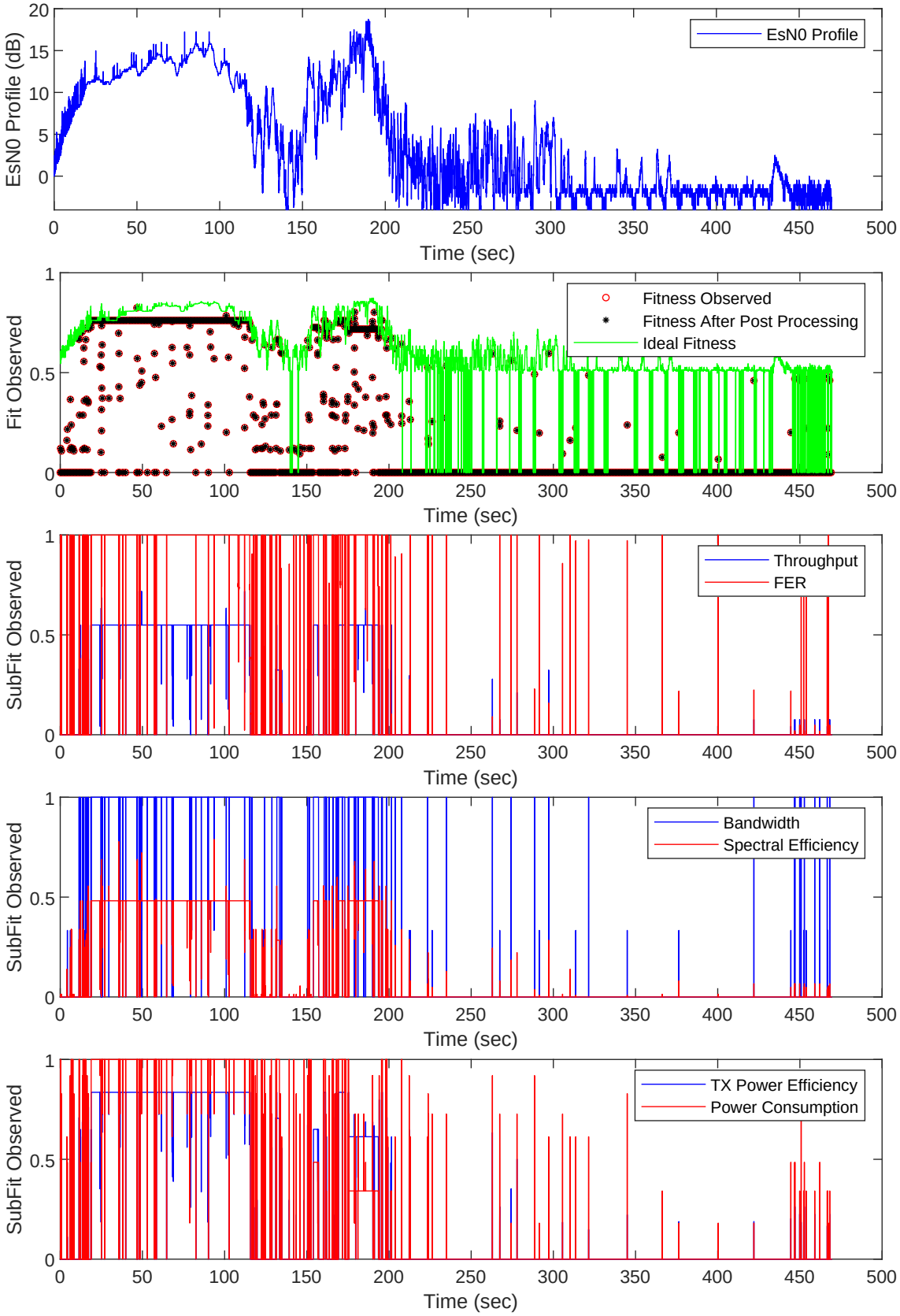


Figure B.32: Operation of CE-RLM on SNR profile 1, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

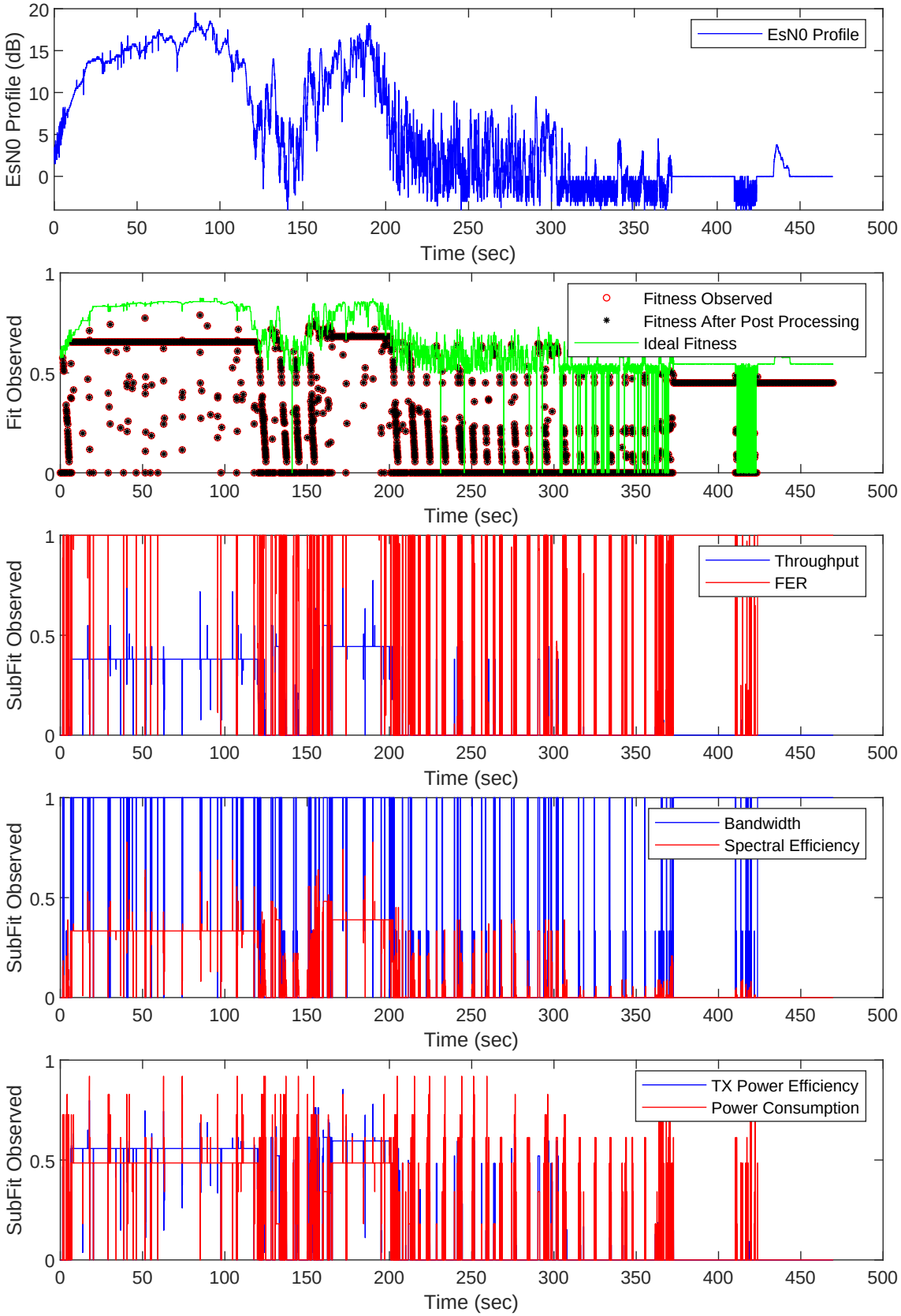


Figure B.33: Operation of CE-NSE on SNR profile 1, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

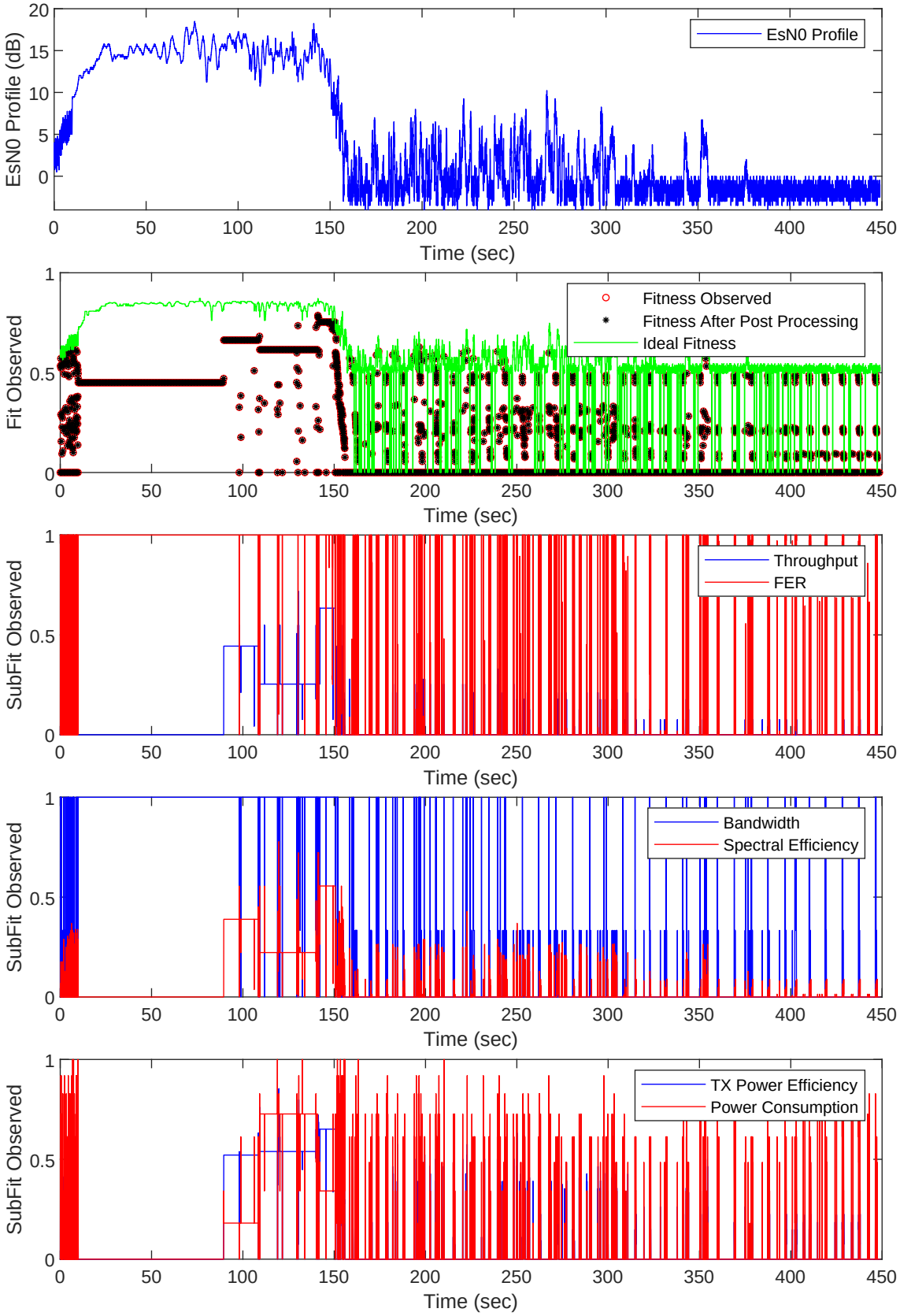


Figure B.34: Operation of CE-LM on SNR profile 2, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

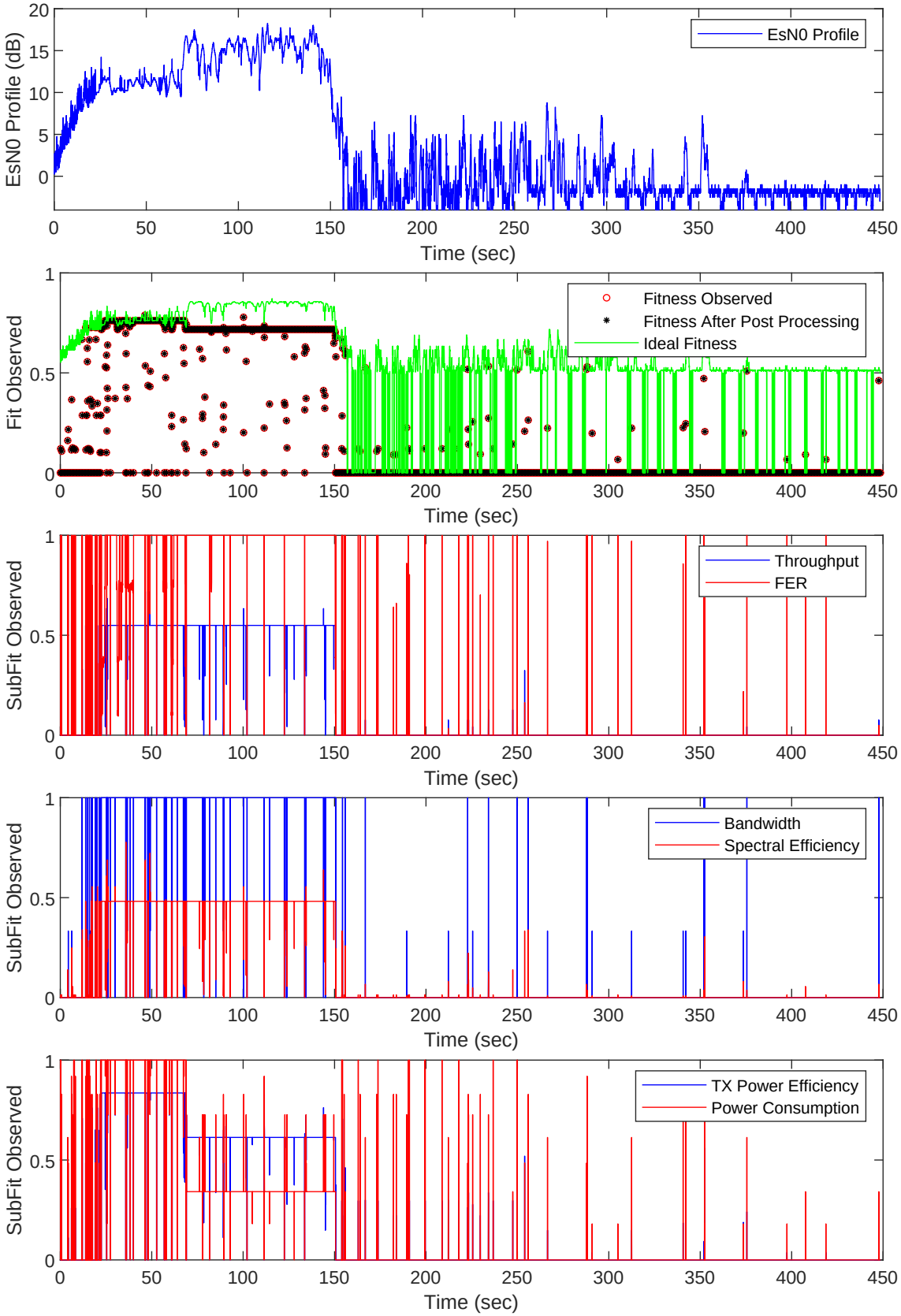


Figure B.35: Operation of CE-RLM on SNR profile 2, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

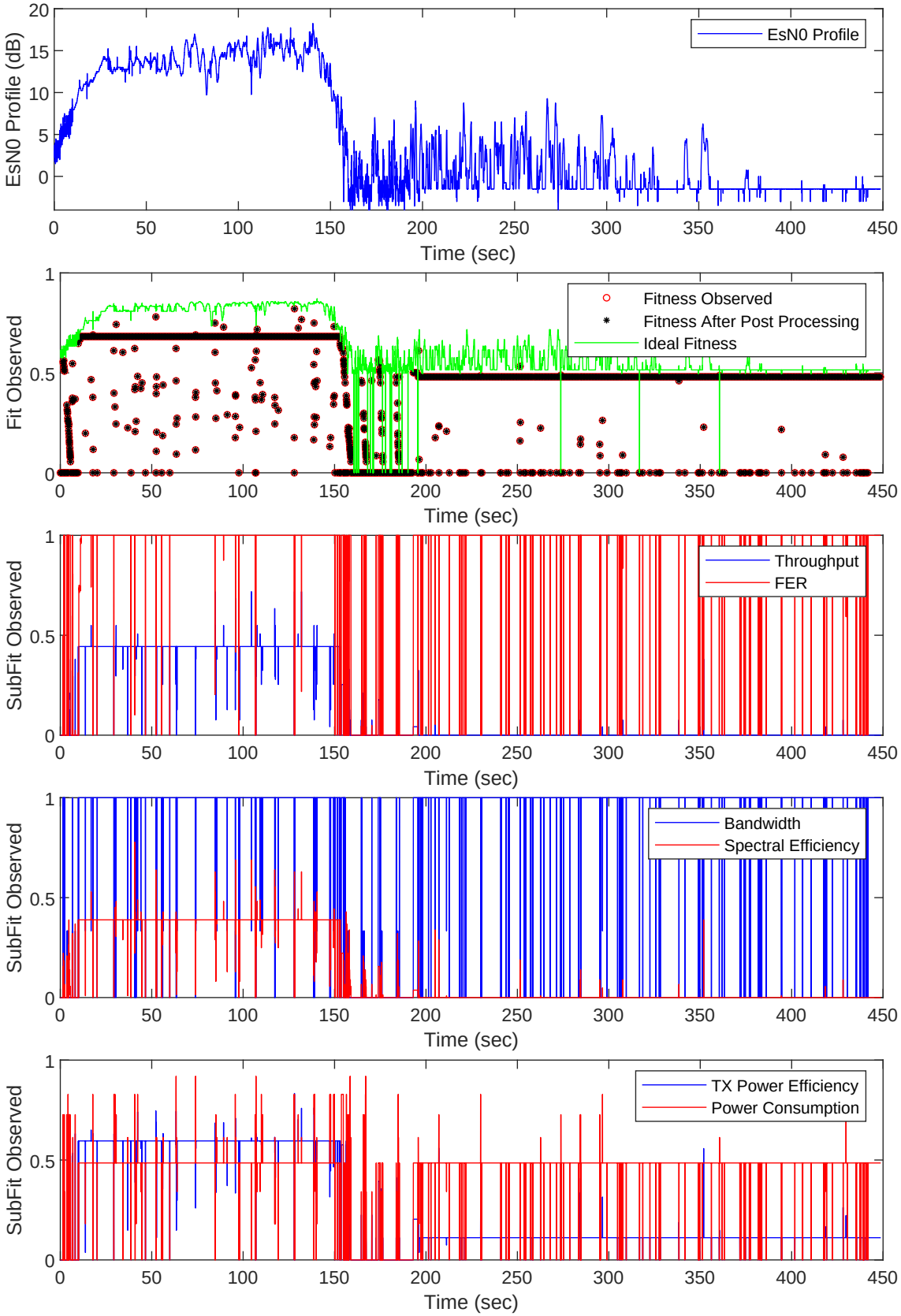


Figure B.36: Operation of CE-NSE on SNR profile 2, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

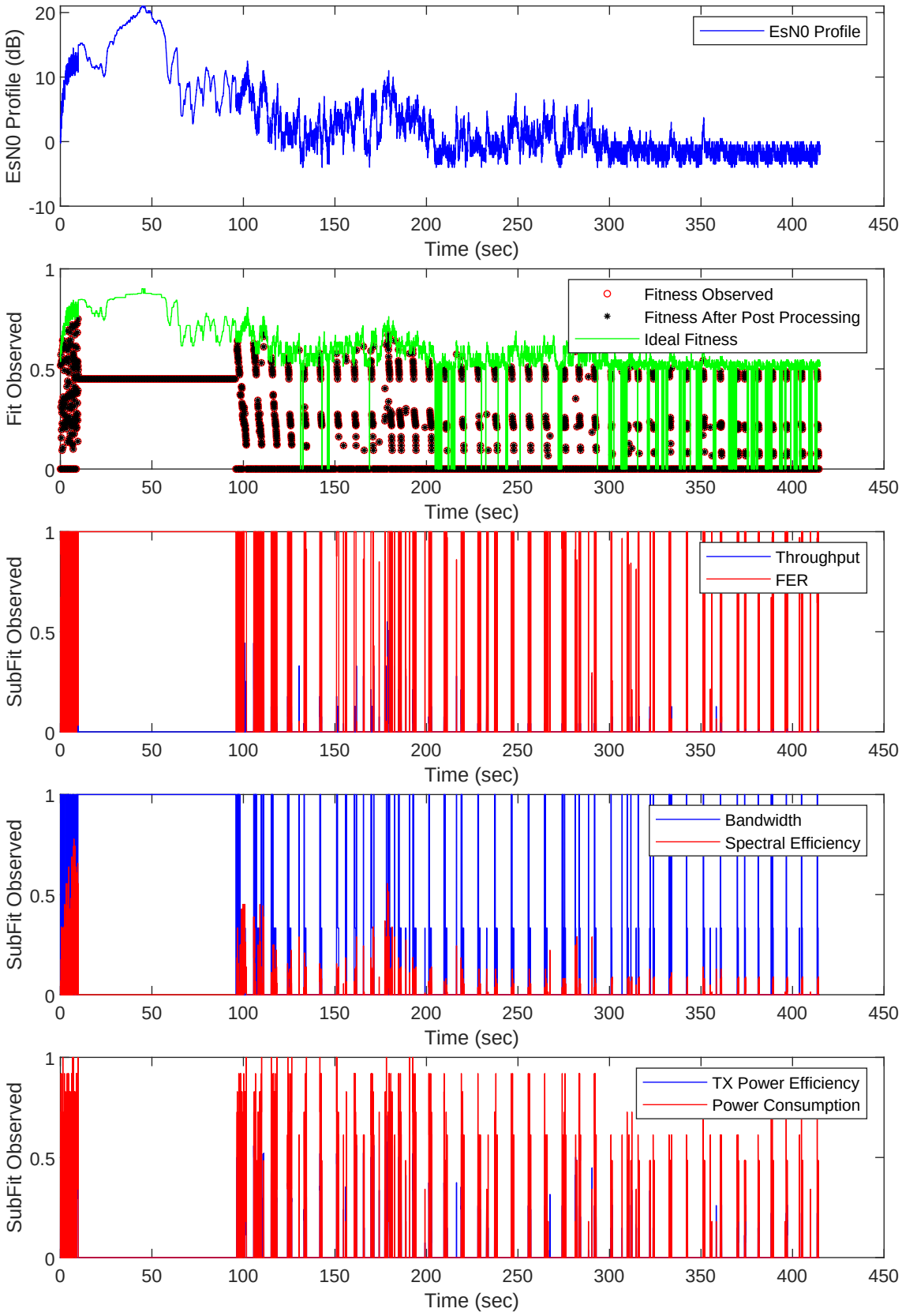


Figure B.37: Operation of CE-LM on SNR profile 3, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

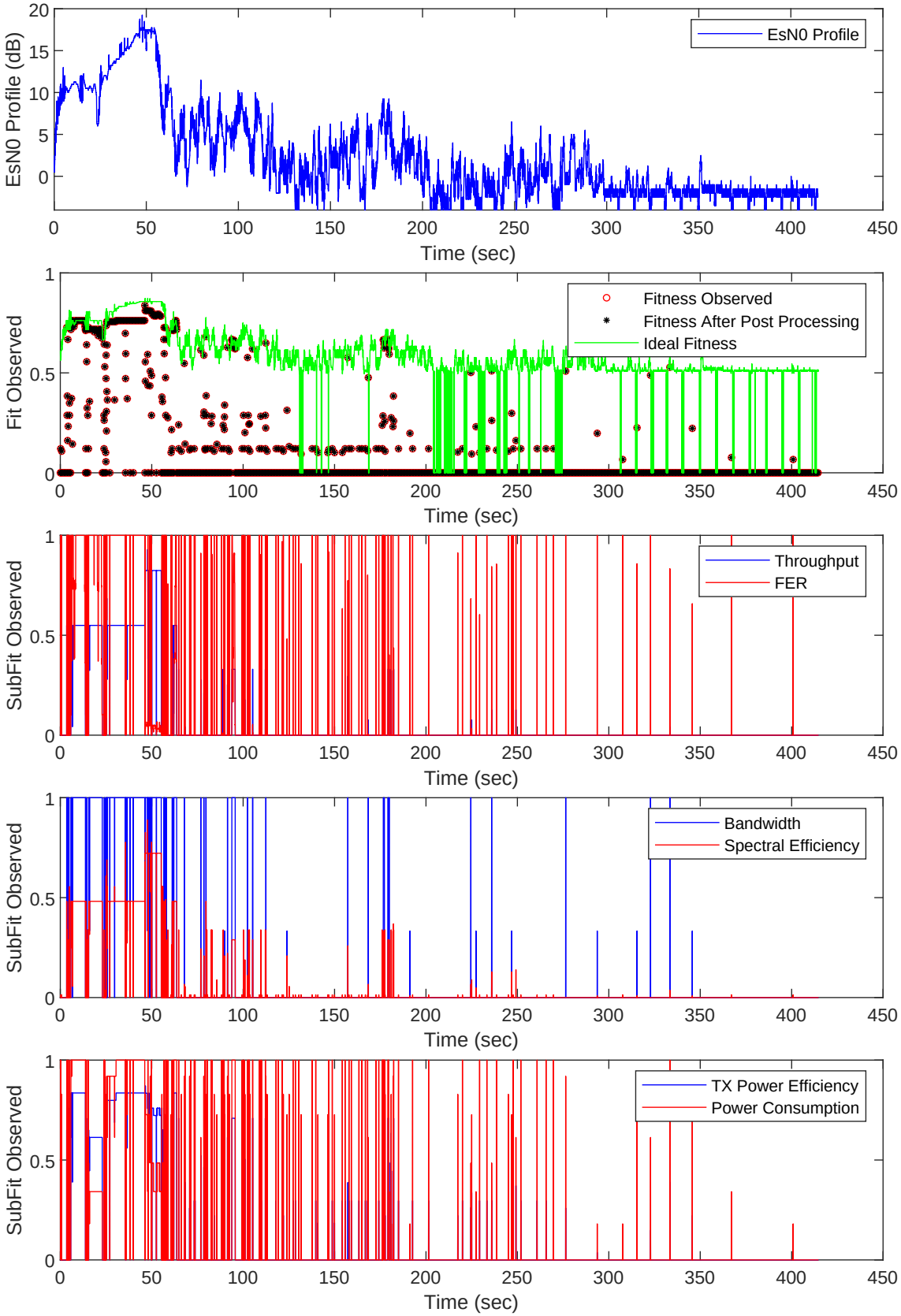


Figure B.38: Operation of CE-RLM on SNR profile 3, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

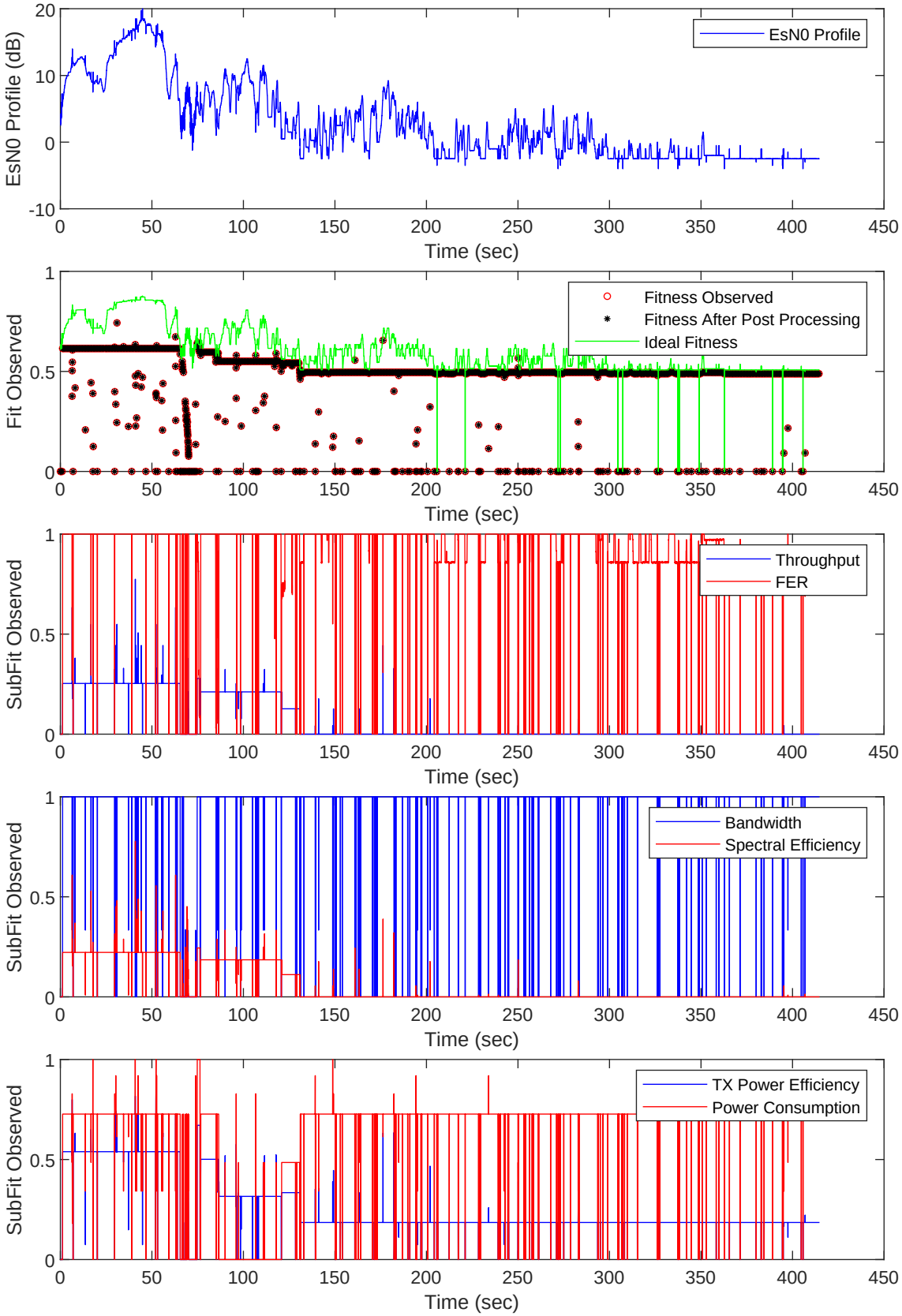


Figure B.39: Operation of CE-NSE on SNR profile 3, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

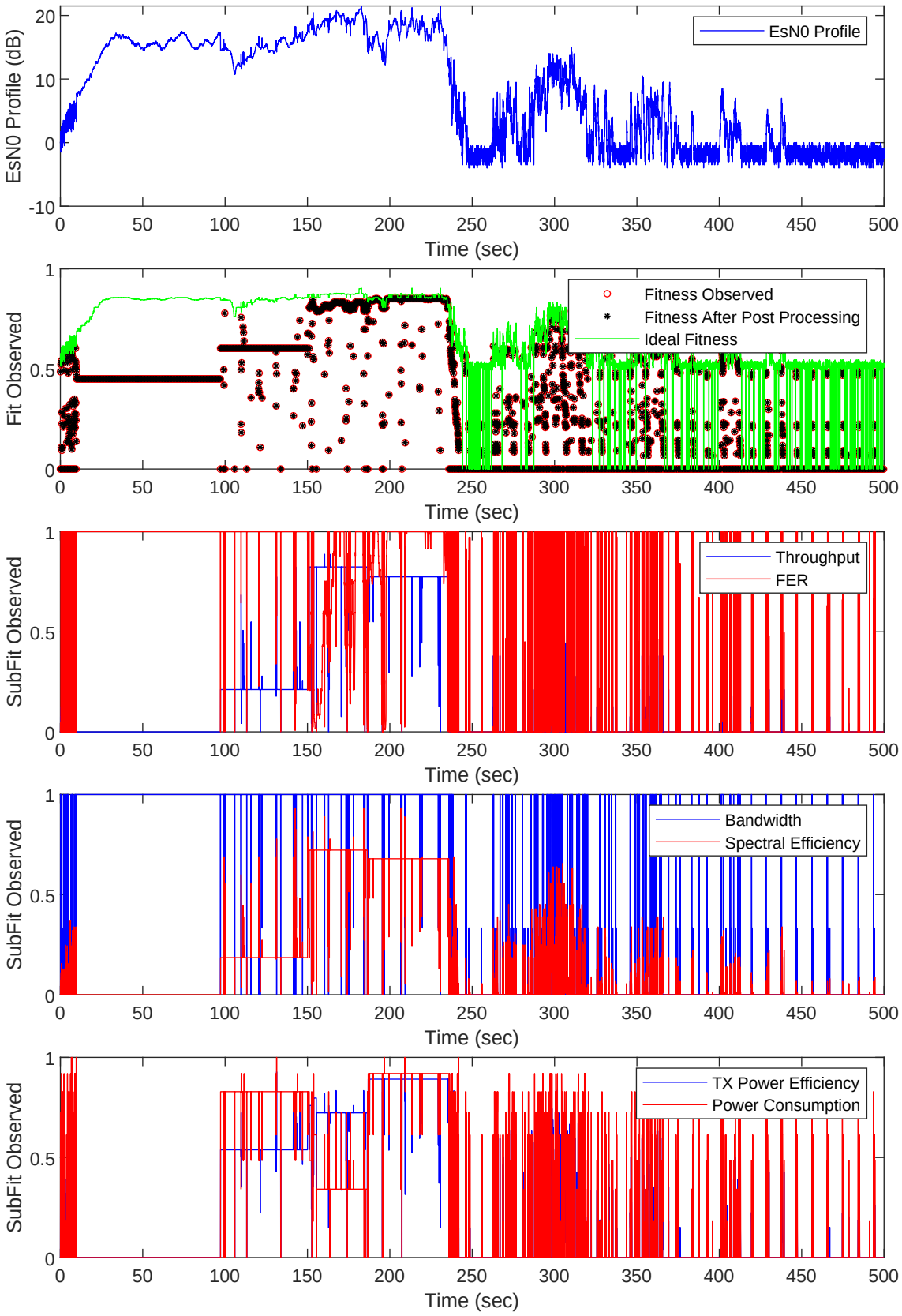


Figure B.40: Operation of CE-LM on SNR profile 4, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

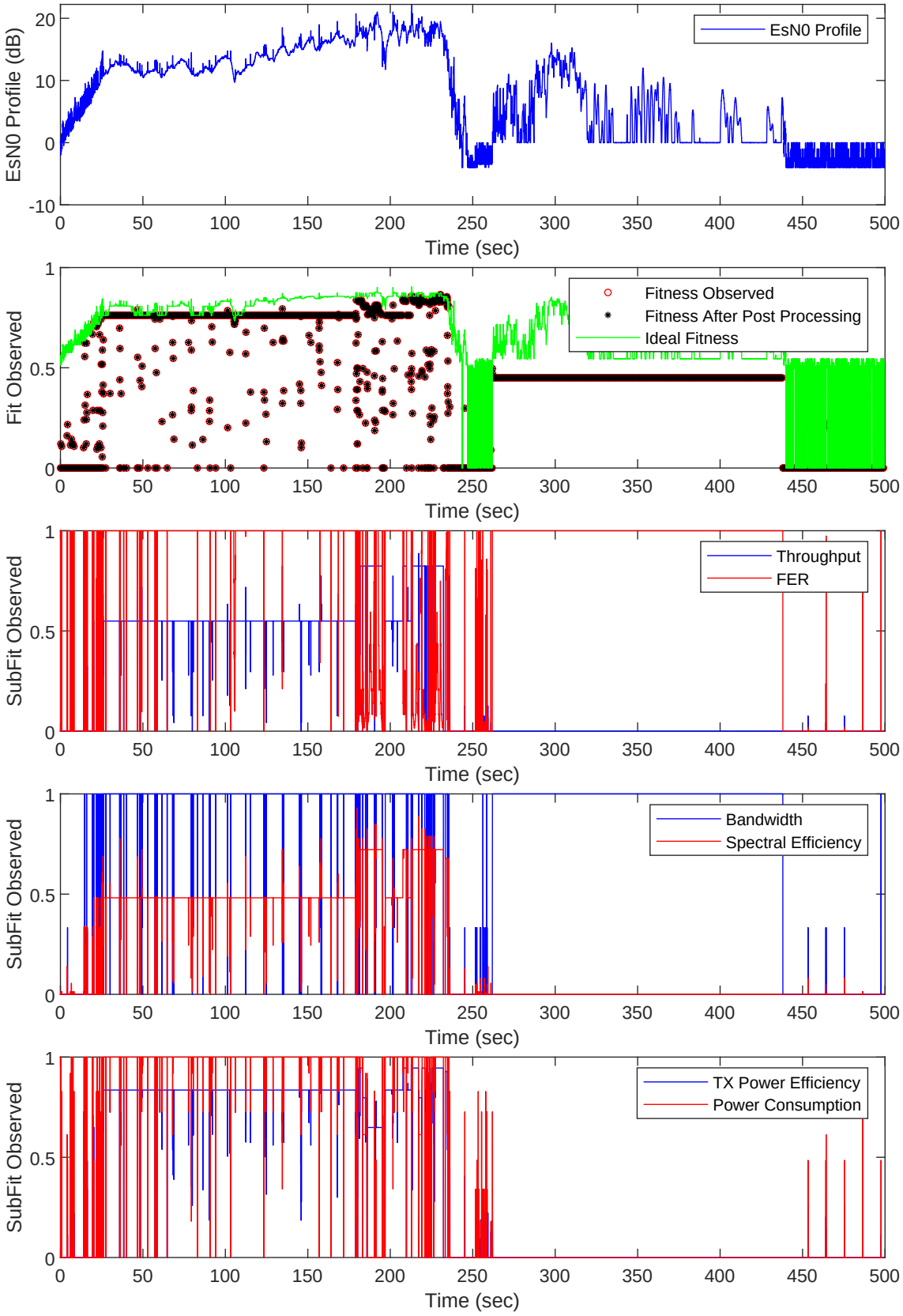


Figure B.41: Operation of CE-RLM on SNR profile 4, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

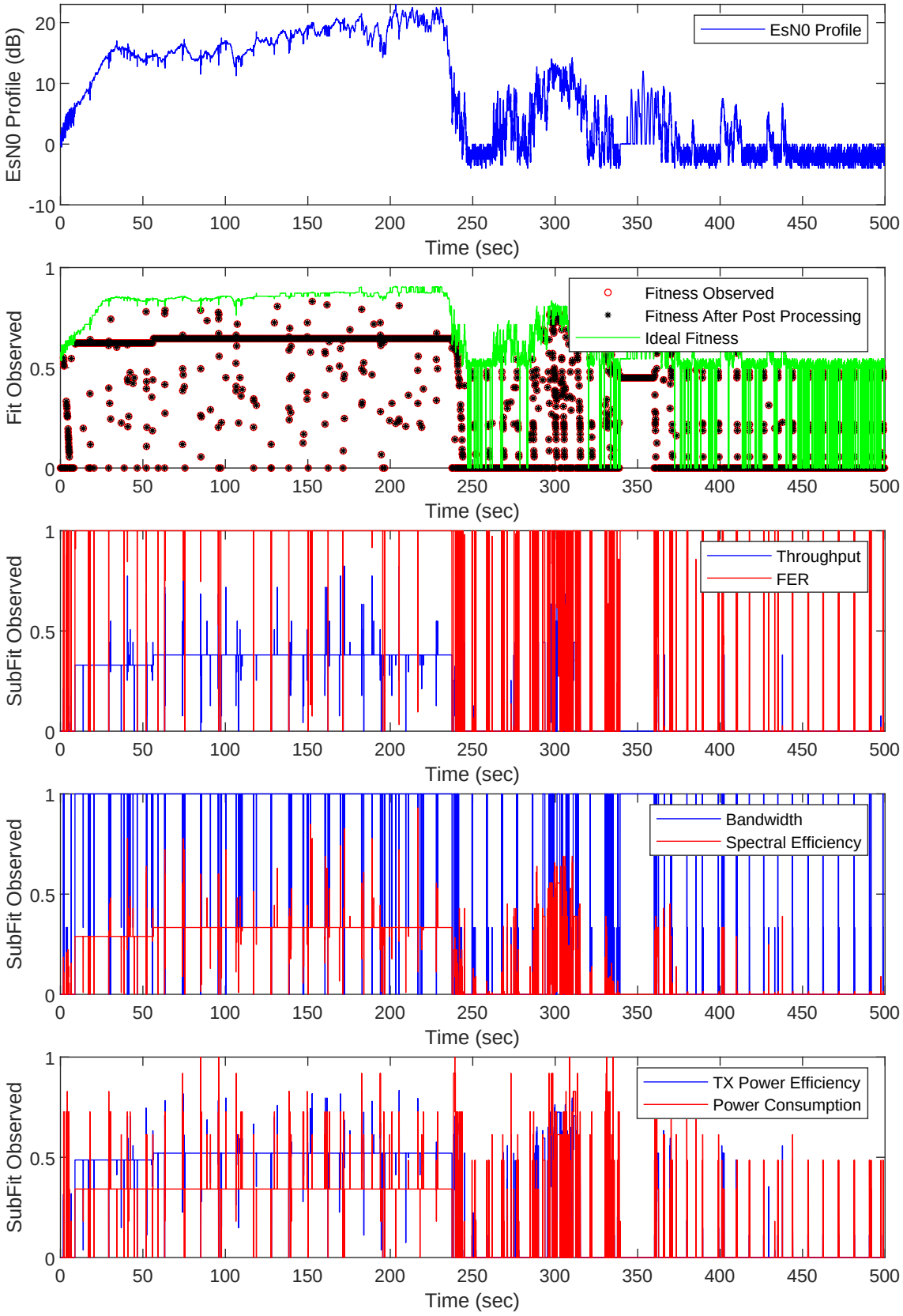


Figure B.42: Operation of CE-NSE on SNR profile 4, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

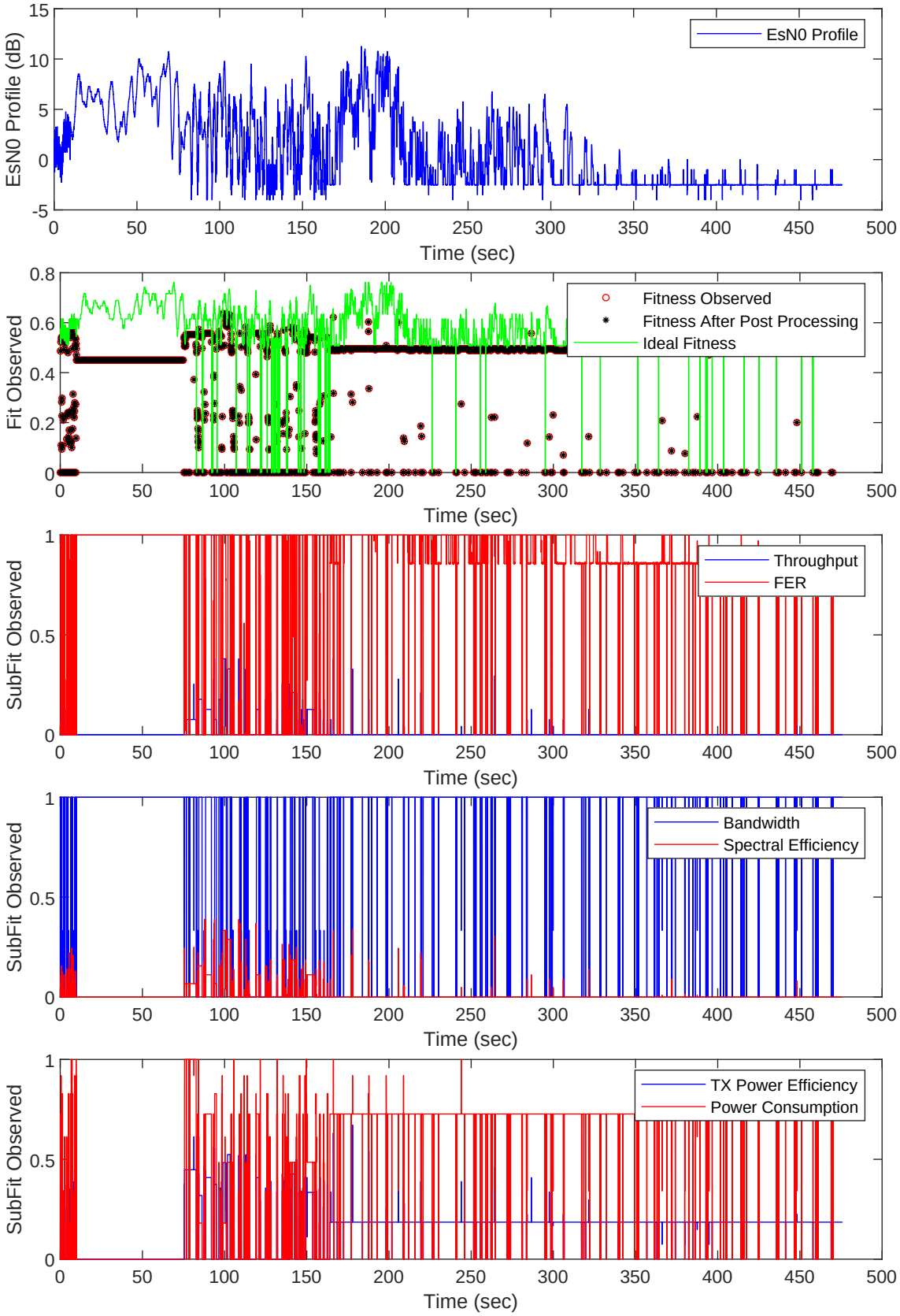


Figure B.43: Operation of CE-LM on SNR profile 5, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

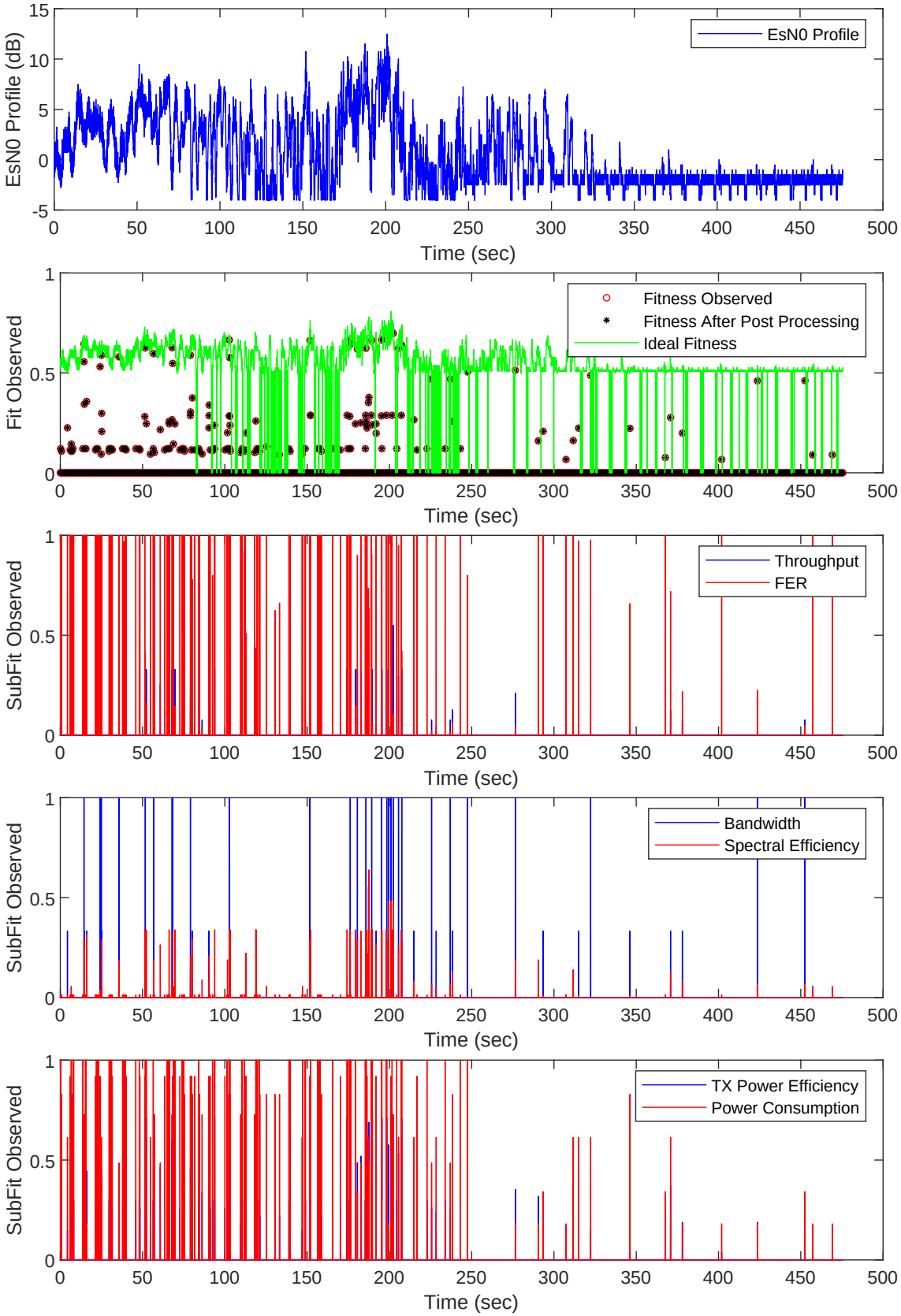


Figure B.44: Operation of CE-RLM on SNR profile 5, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

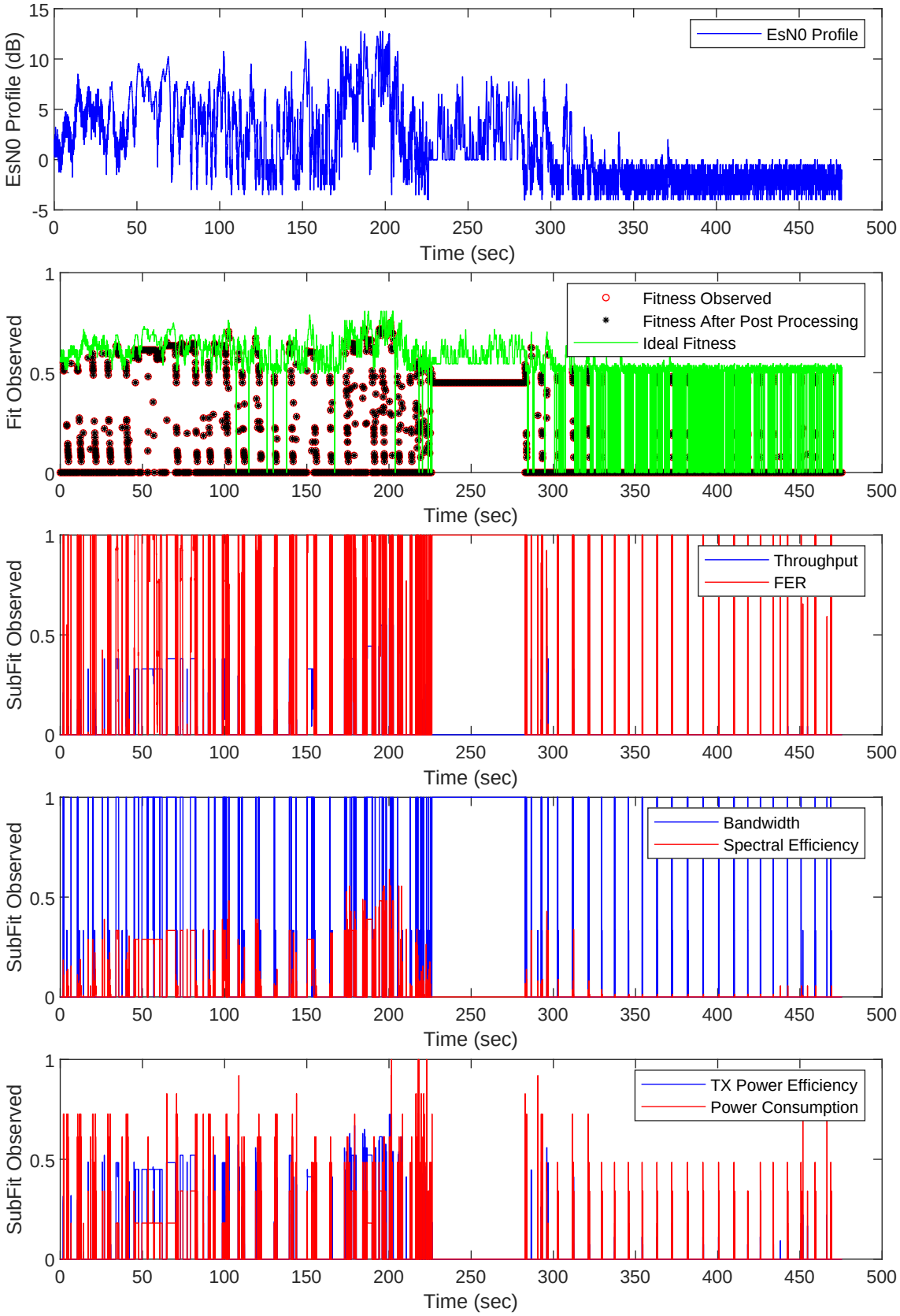
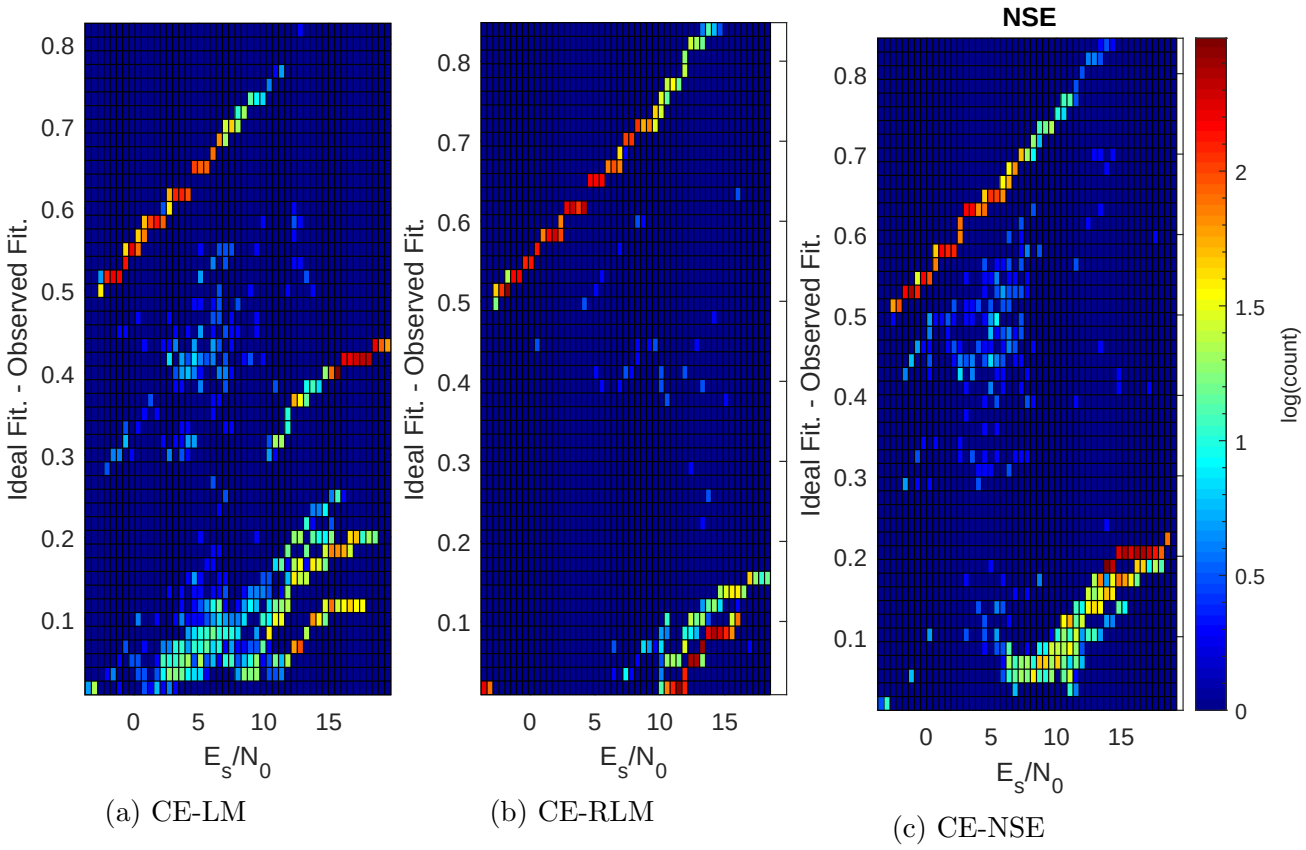


Figure B.45: Operation of CE-NSE on SNR profile 5, using the Emergency mission. Includes SNR profile, fitness observed, and subfitness values observed.

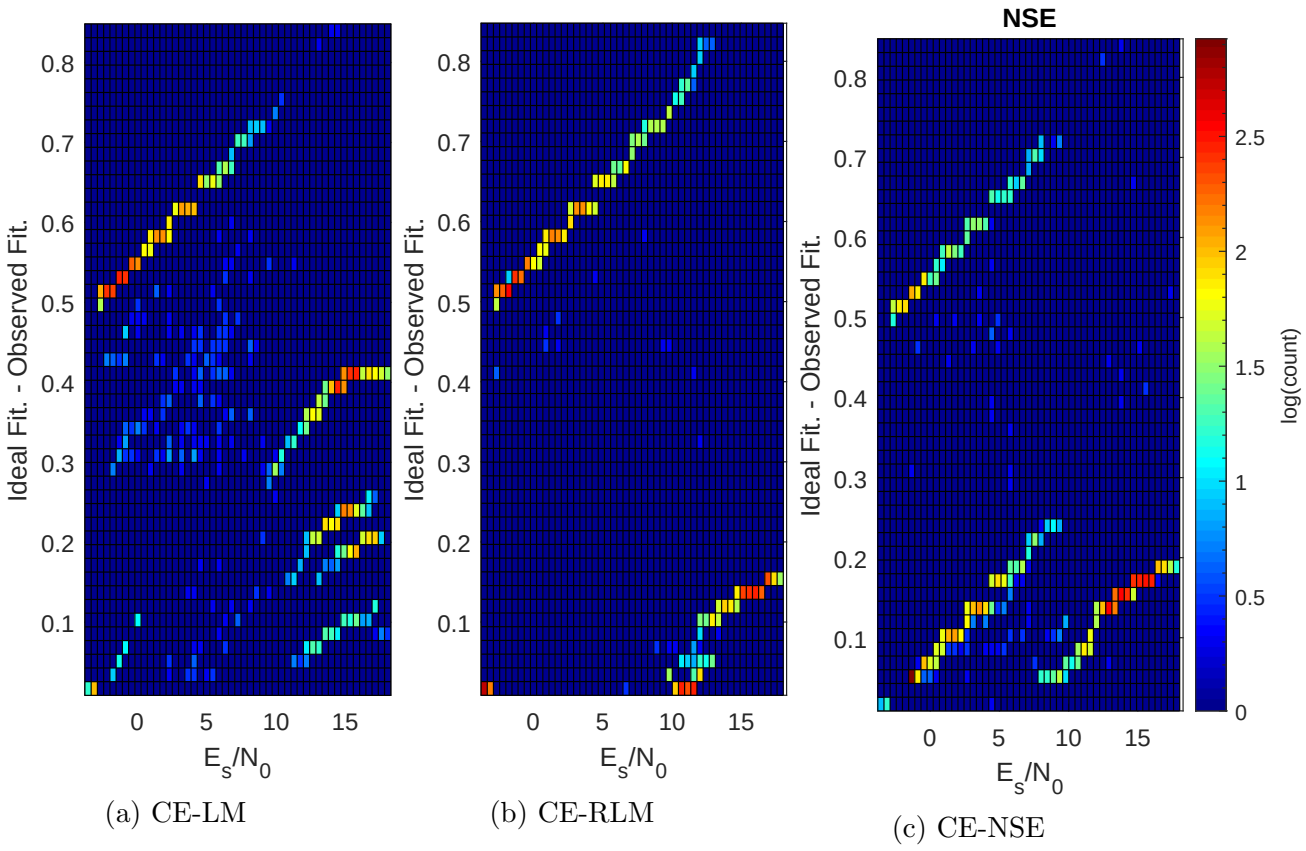
B.2 C Simulation 2D Histograms

B.2.1 Cooperation Mission



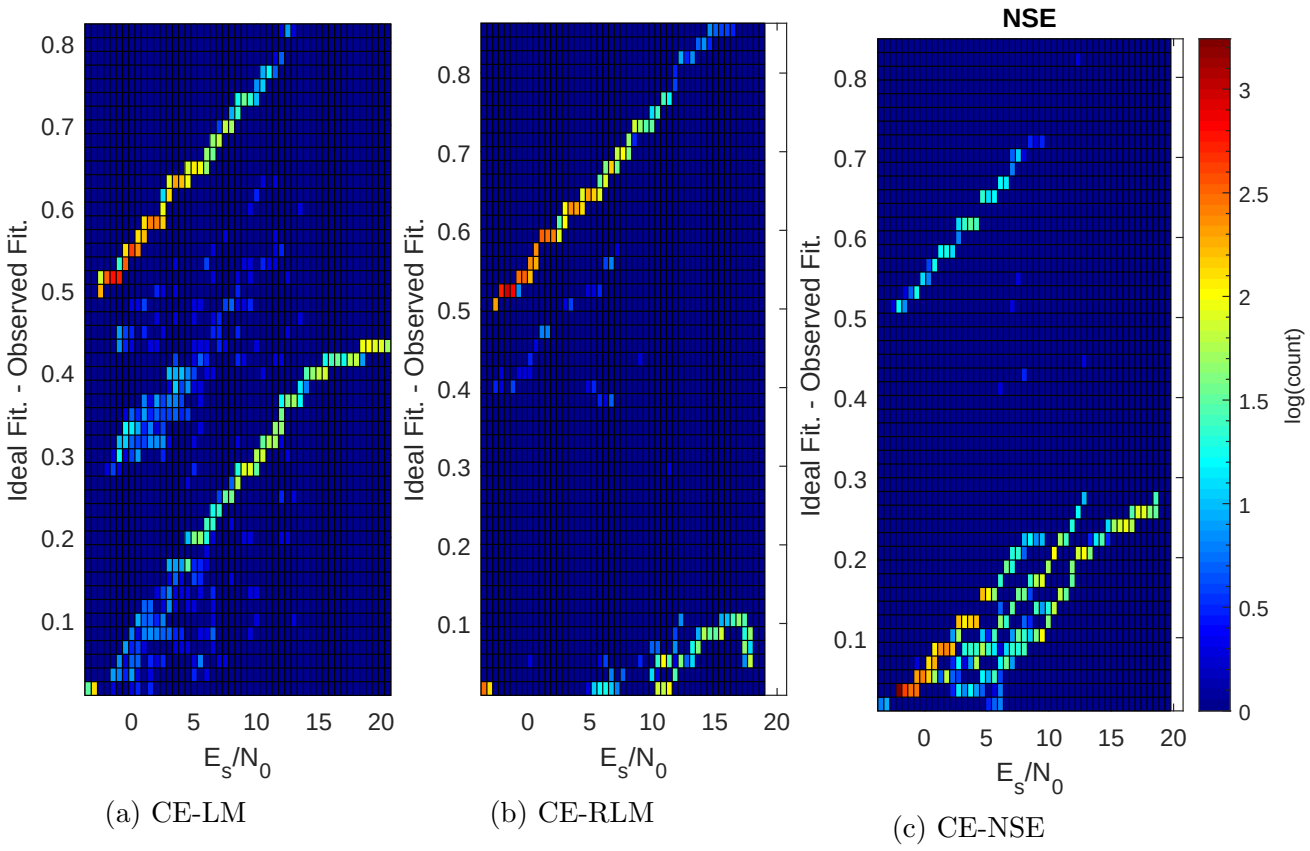
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure B.46: Two-dimensional histograms of each training method operating the Cooperation mission on SNR profile 1. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



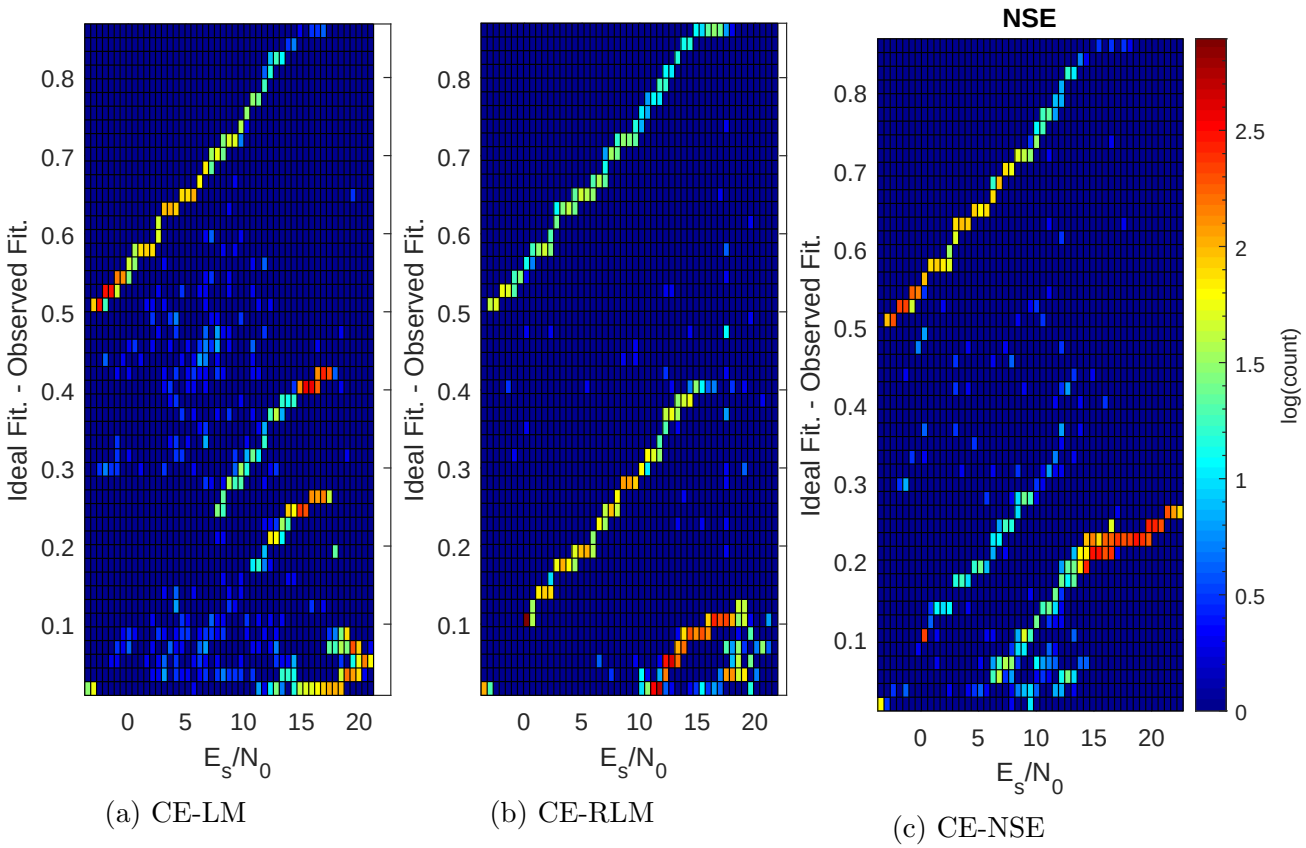
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure B.47: Two-dimensional histograms of each training method operating the Cooperation mission on SNR profile 2. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



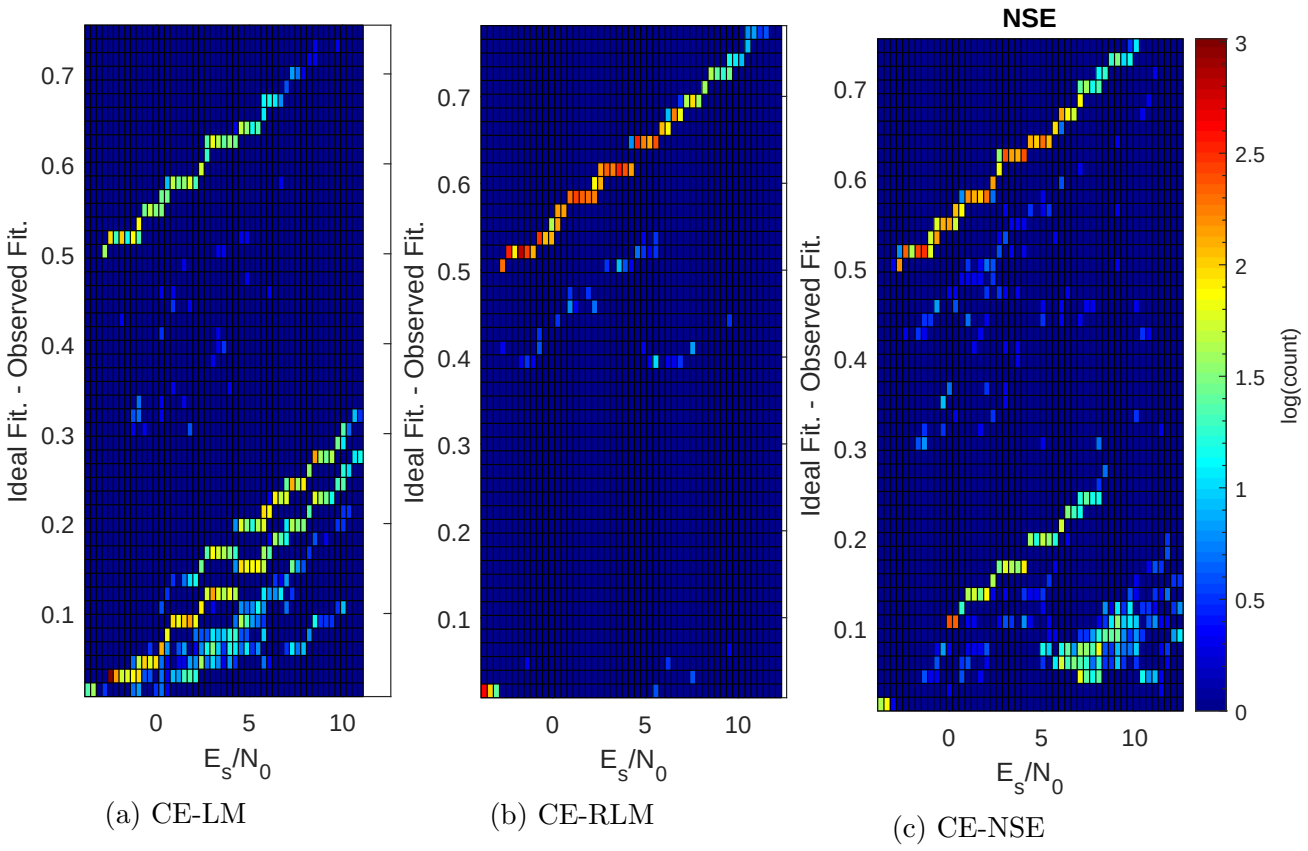
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure B.48: Two-dimensional histograms of each training method operating the Cooperation mission on SNR profile 3. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



(a) CE-LM (b) CE-RLM (c) CE-NSE

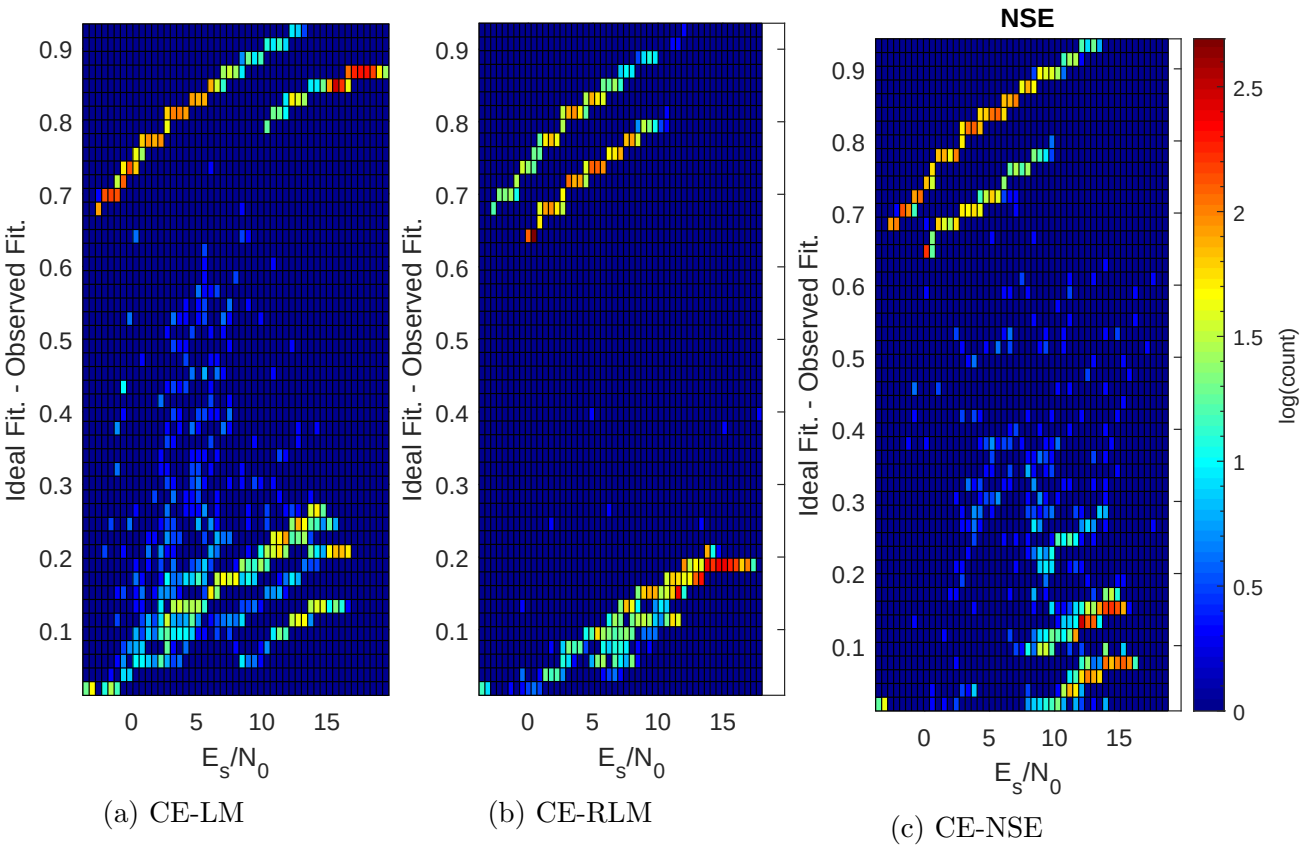
Figure B.49: Two-dimensional histograms of each training method operating the Cooperation mission on SNR profile 4. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



(a) CE-LM (b) CE-RLM (c) CE-NSE

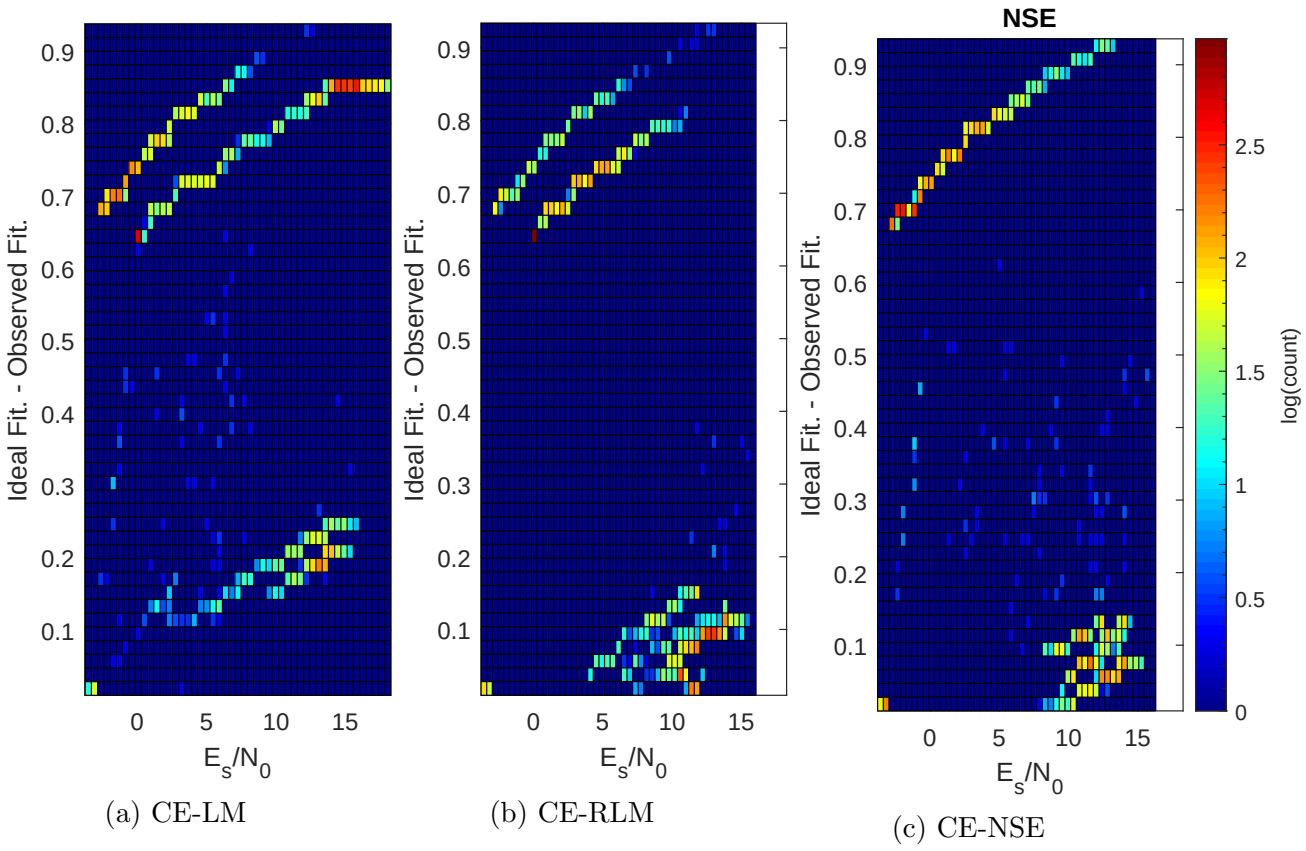
Figure B.50: Two-dimensional histograms of each training method operating the Cooperation mission on SNR profile 5. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)

B.2.2 Power Saving Mission



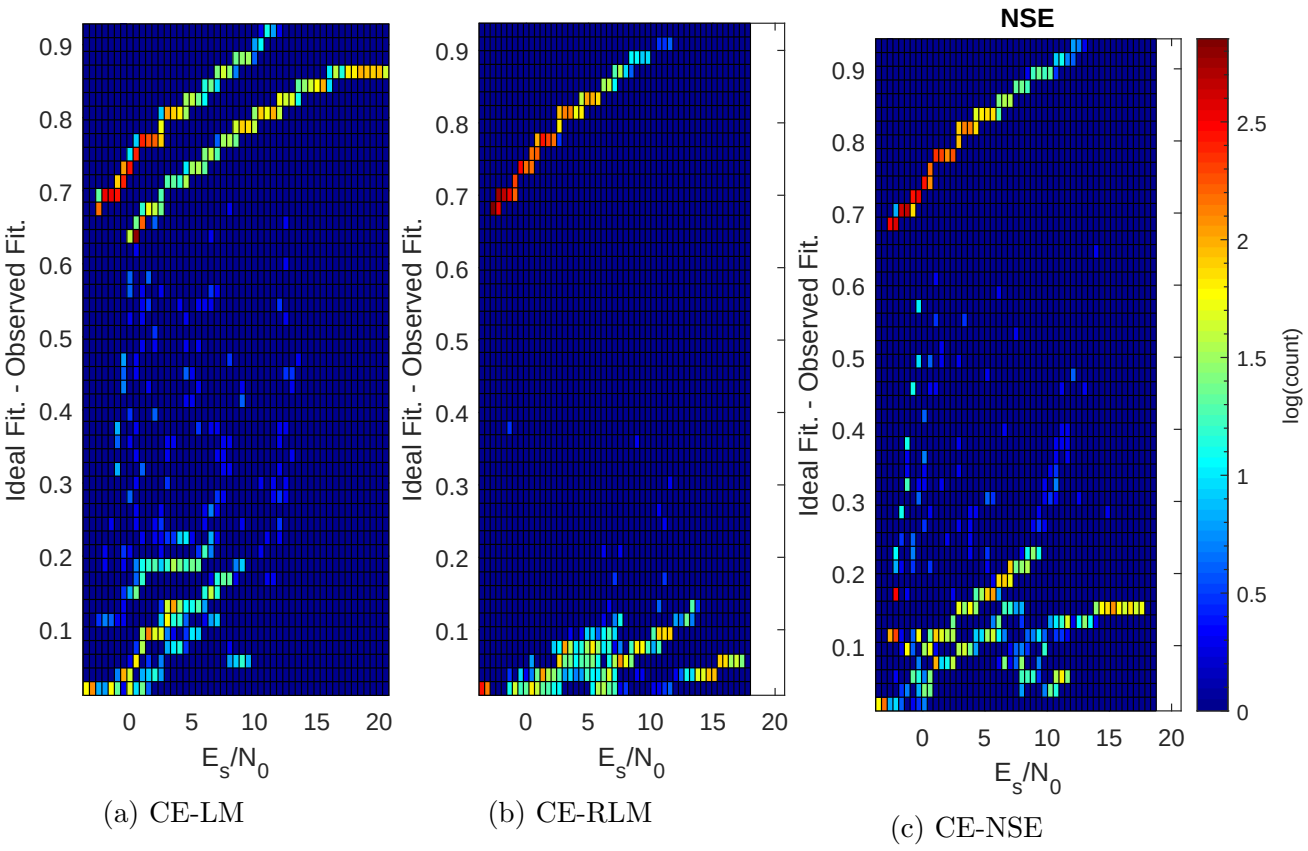
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure B.51: Two-dimensional histograms of each training method operating the Power Saving mission on SNR profile 1. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



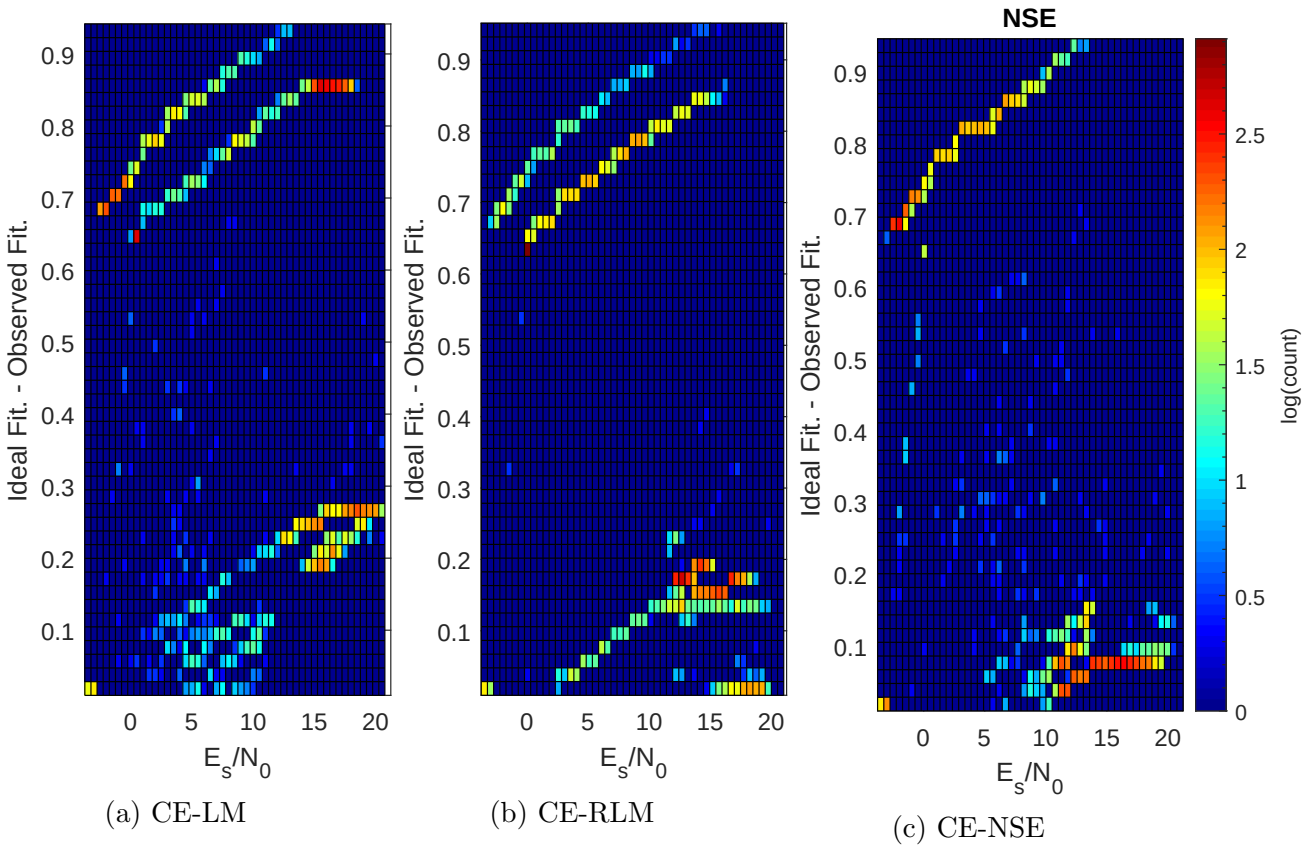
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure B.52: Two-dimensional histograms of each training method operating the Power Saving mission on SNR profile 2. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



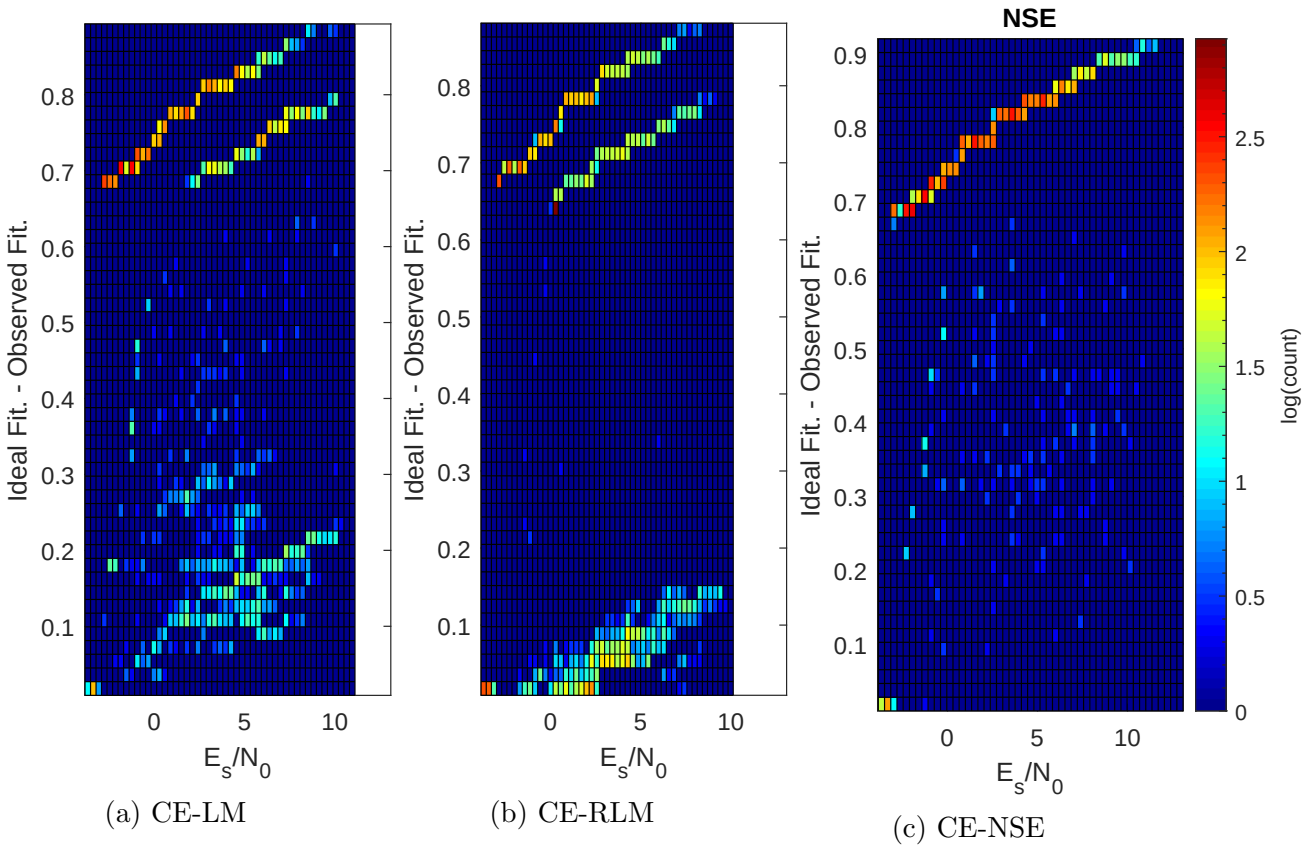
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure B.53: Two-dimensional histograms of each training method operating the Power Saving mission on SNR profile 3. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



(a) CE-LM (b) CE-RLM (c) CE-NSE

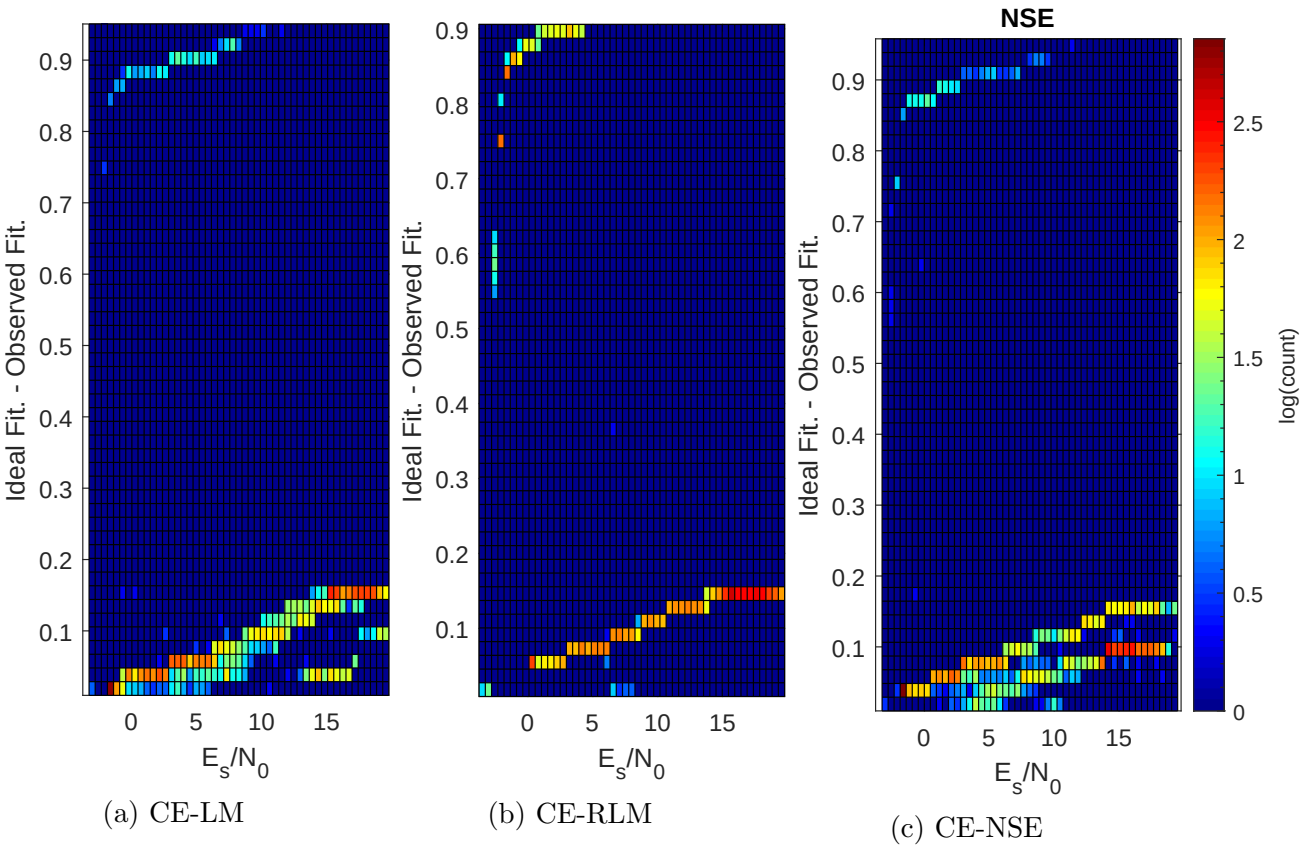
Figure B.54: Two-dimensional histograms of each training method operating the Power Saving mission on SNR profile 4. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



(a) CE-LM (b) CE-RLM (c) CE-NSE

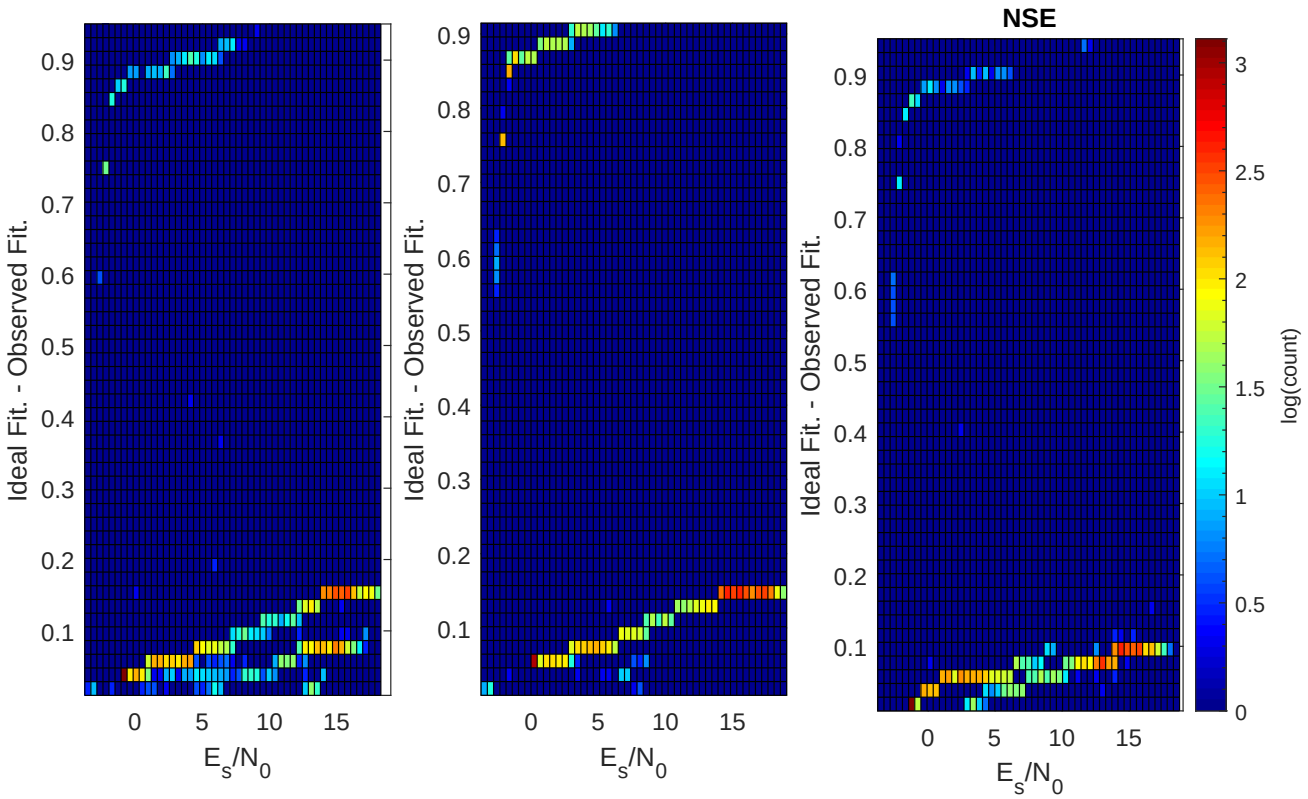
Figure B.55: Two-dimensional histograms of each training method operating the Power Saving mission on SNR profile 5. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)

B.2.3 Emergency Mission



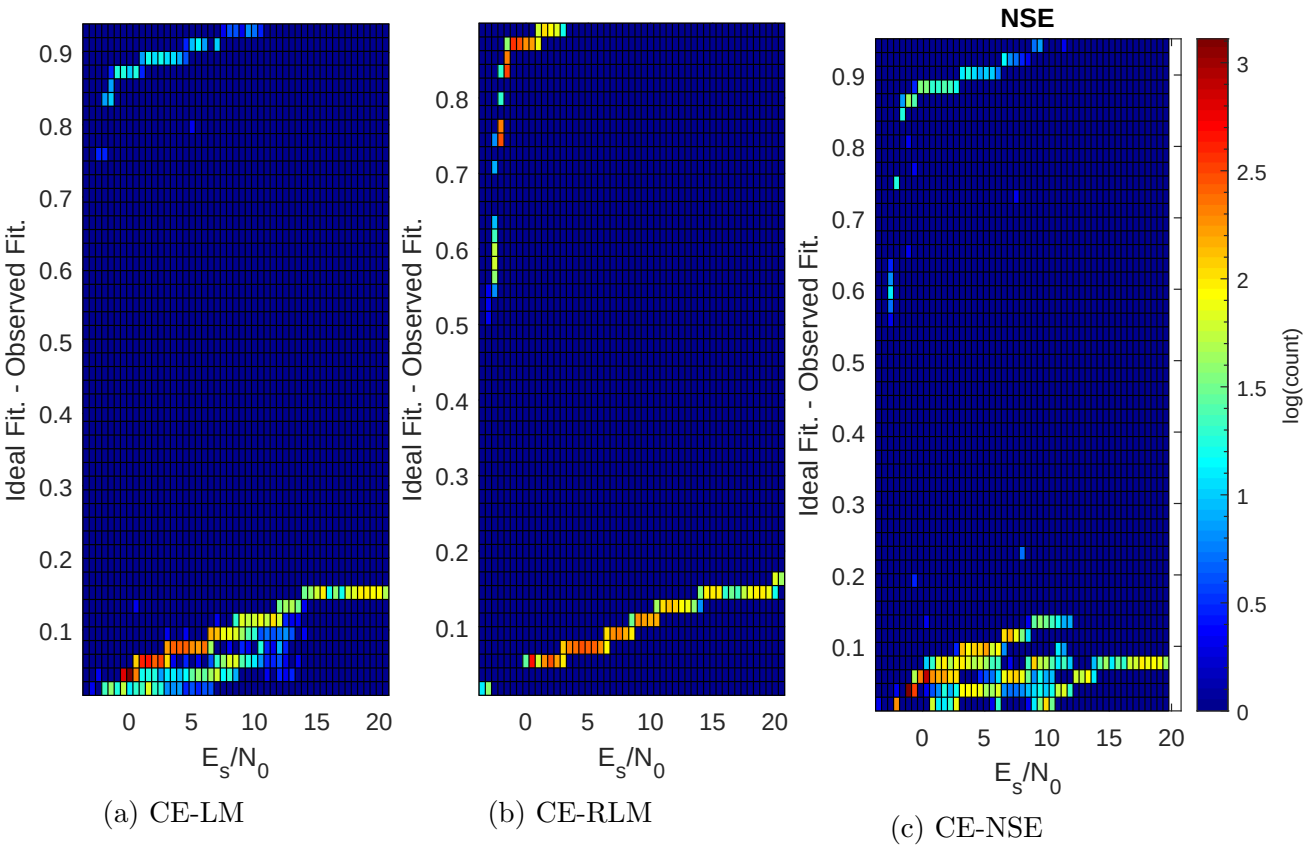
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure B.56: Two-dimensional histograms of each training method operating the Emergency mission on SNR profile 1. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



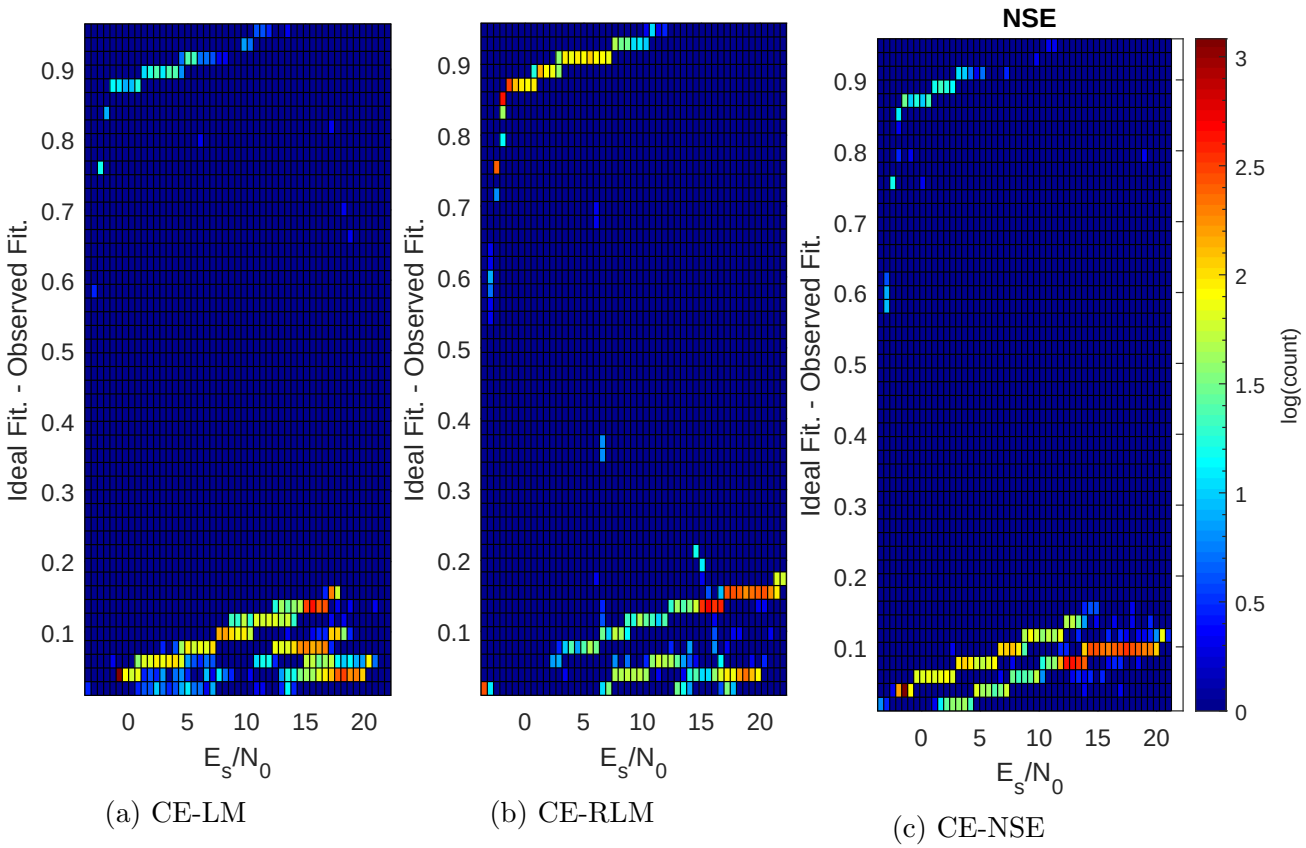
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure B.57: Two-dimensional histograms of each training method operating the Emergency mission on SNR profile 2. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



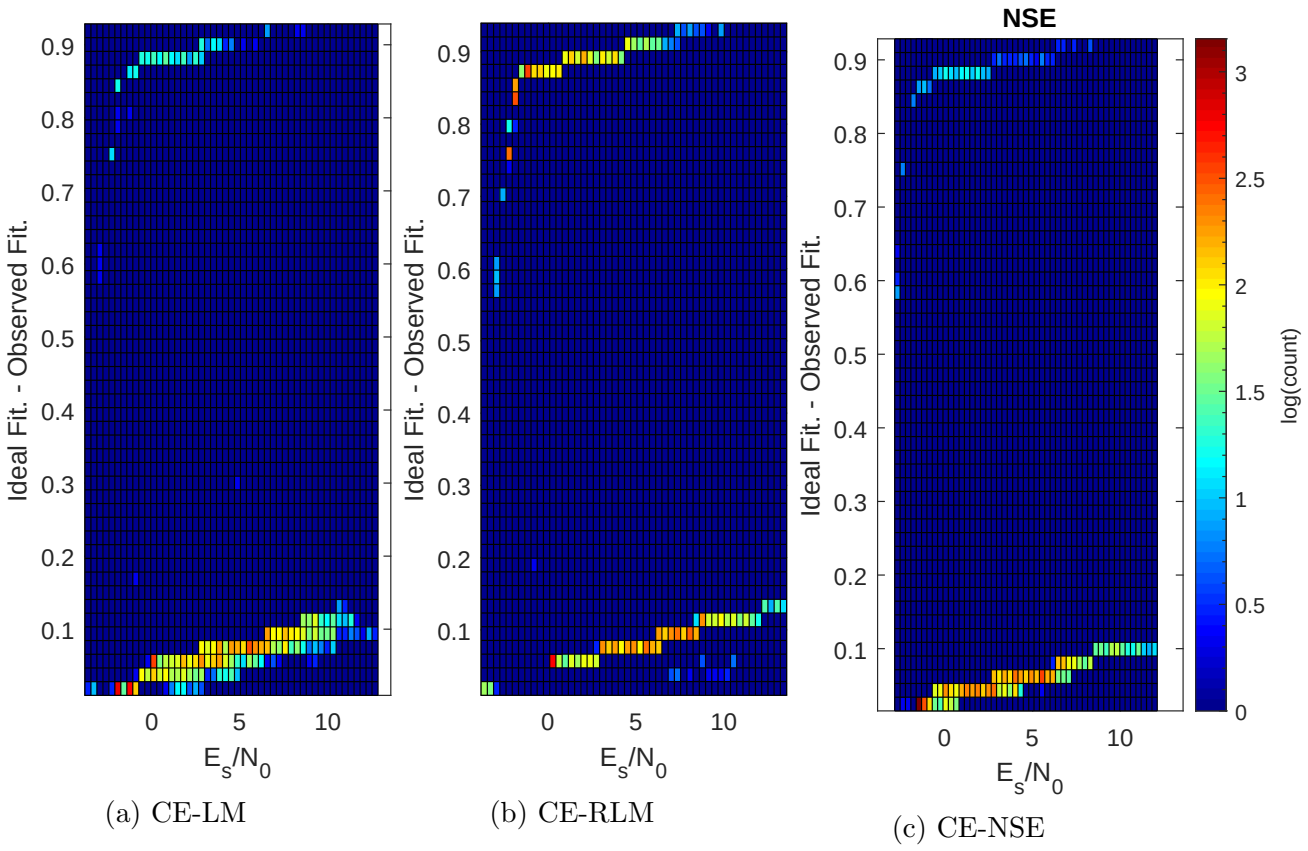
(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure B.58: Two-dimensional histograms of each training method operating the Emergency mission on SNR profile 3. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)



(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure B.59: Two-dimensional histograms of each training method operating the Emergency mission on SNR profile 4. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)

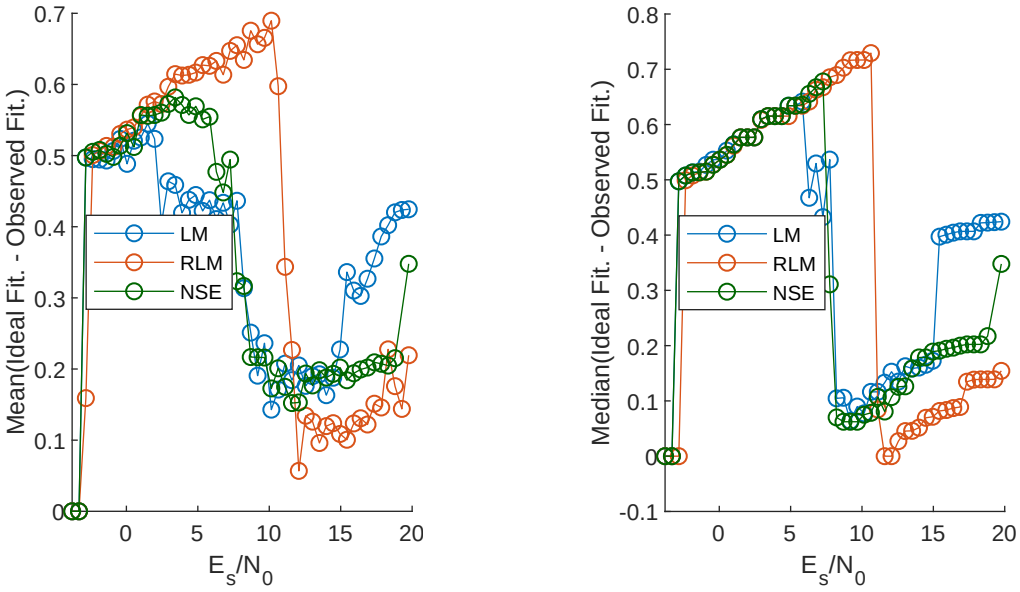


(a) CE-LM (b) CE-RLM (c) CE-NSE

Figure B.60: Two-dimensional histograms of each training method operating the Emergency mission on SNR profile 5. The dimensions are (E_s/N_0, fitness score, log_10(number of frames observed)

B.3 C++ Simulation Binned Mean Plots

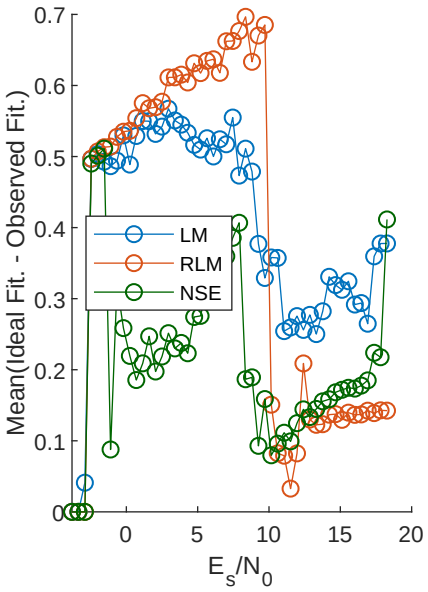
B.3.1 Cooperation Mission



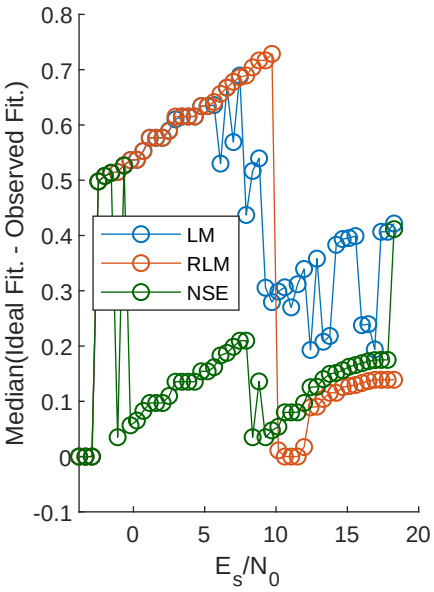
(a) Binned mean of fitness distance between ideal fitness and observed fitness.

(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.61: Binned mean and binned median plots for the different CE training methods running the Cooperation mission on SNR profile 1. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

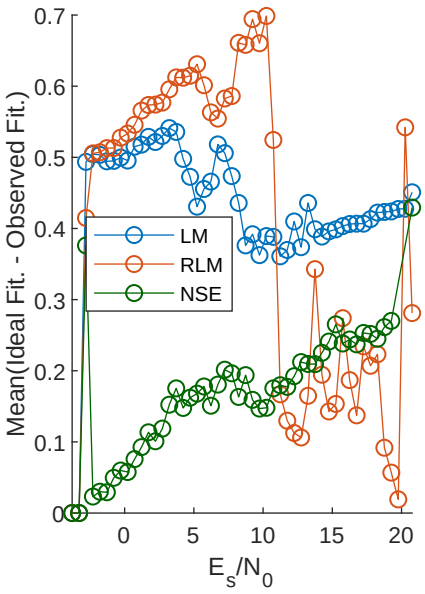


(a) Binned mean of fitness distance between ideal fitness and observed fitness.

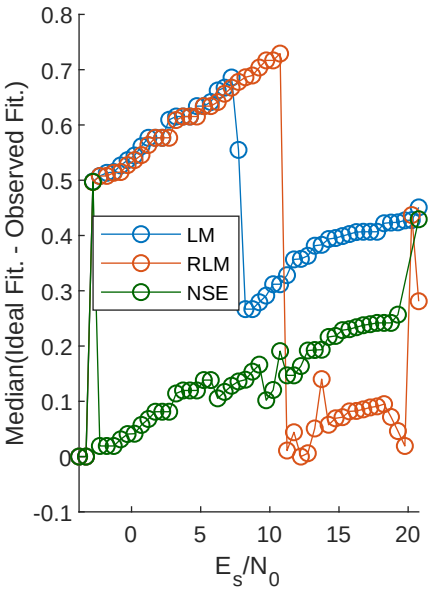


(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.62: Binned mean and binned median plots for the different CE training methods running the Cooperation mission on SNR profile 2. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

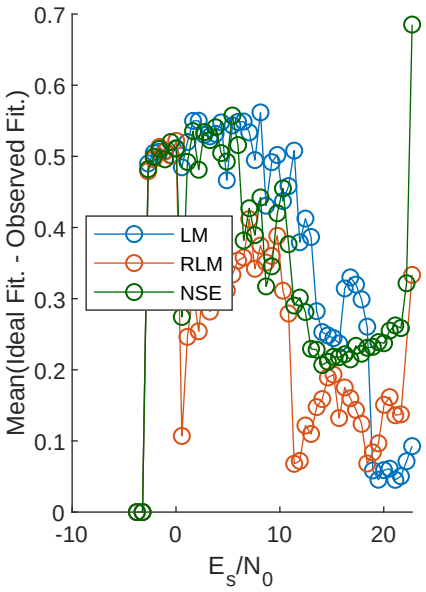


(a) Binned mean of fitness distance between ideal fitness and observed fitness.

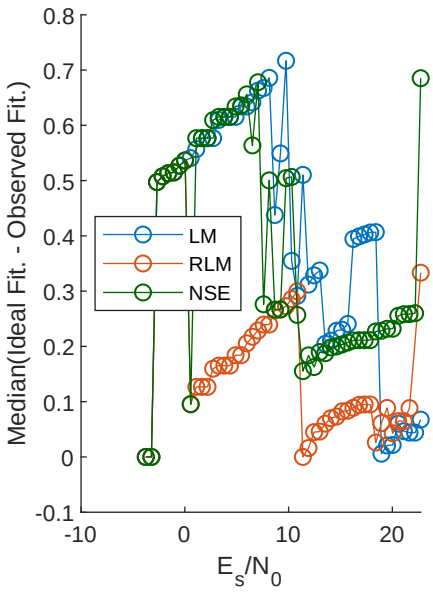


(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.63: Binned mean and binned median plots for the different CE training methods running the Cooperation mission on SNR profile 3. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

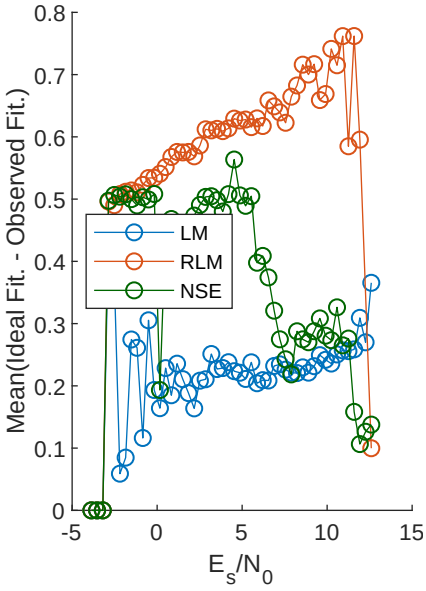


(a) Binned mean of fitness distance between ideal fitness and observed fitness.

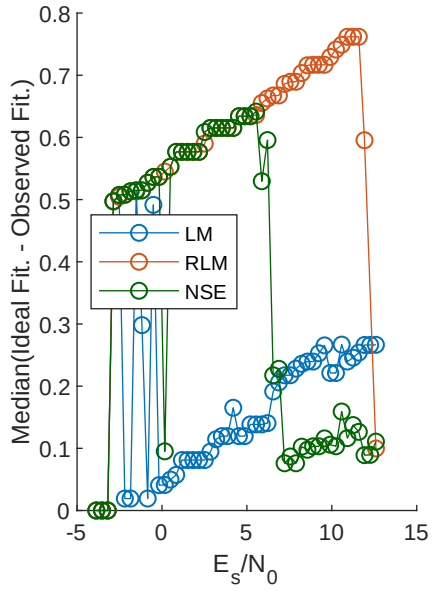


(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.64: Binned mean and binned median plots for the different CE training methods running the Cooperation mission on SNR profile 4. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.



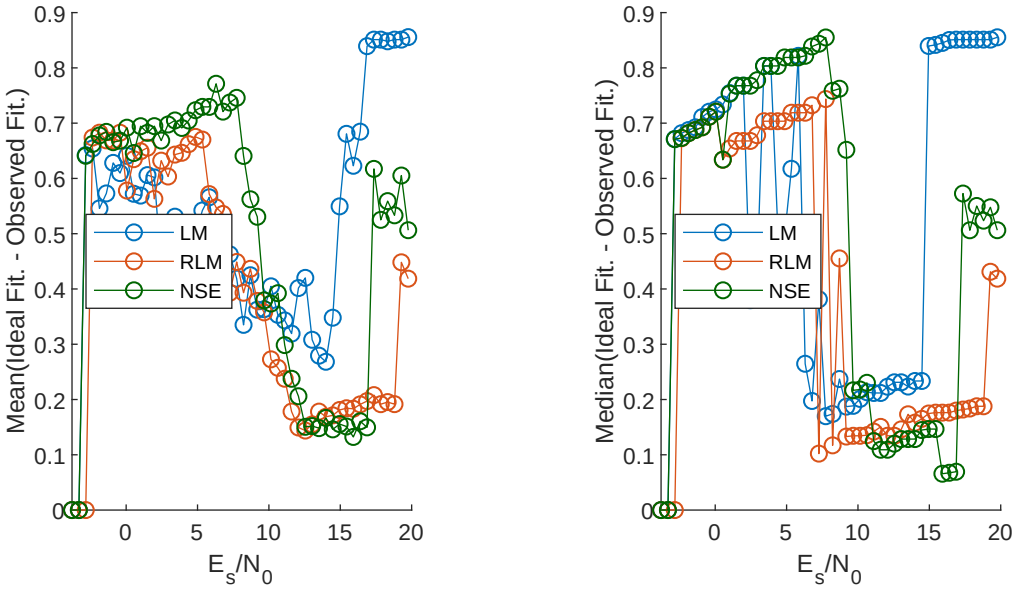
(a) Binned mean of fitness distance between ideal fitness and observed fitness.



(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.65: Binned mean and binned median plots for the different CE training methods running the Cooperation mission on SNR profile 5. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

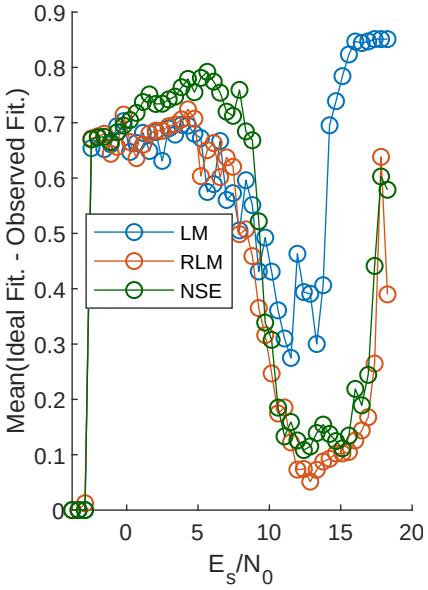
B.3.2 Power Saving Mission



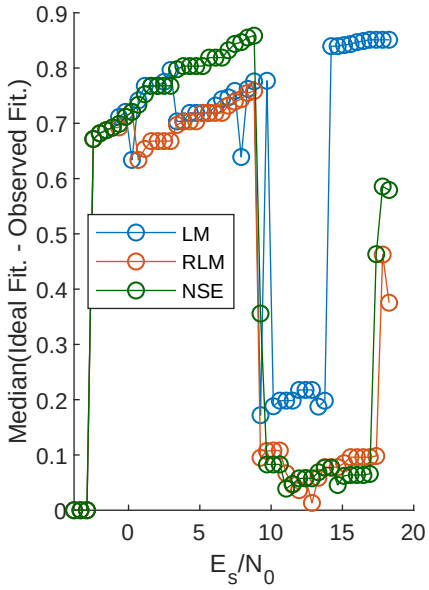
(a) Binned mean of fitness distance between ideal fitness and observed fitness.

(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.66: Binned mean and binned median plots for the different CE training methods running the Power Saving mission on SNR profile 1. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

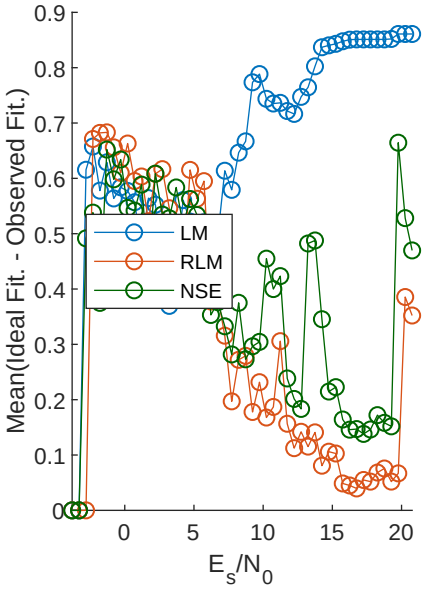


(a) Binned mean of fitness distance between ideal fitness and observed fitness.

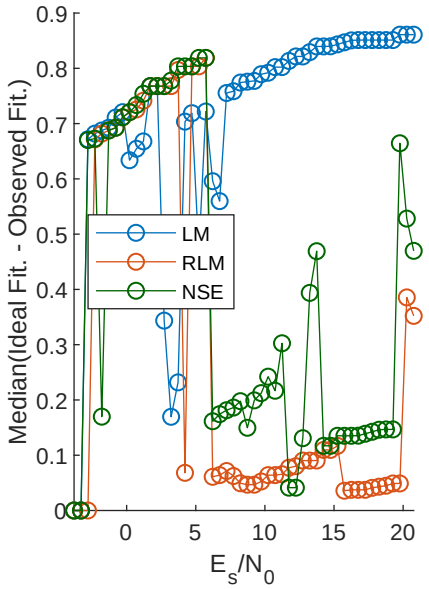


(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.67: Binned mean and binned median plots for the different CE training methods running the Power Saving mission on SNR profile 2. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

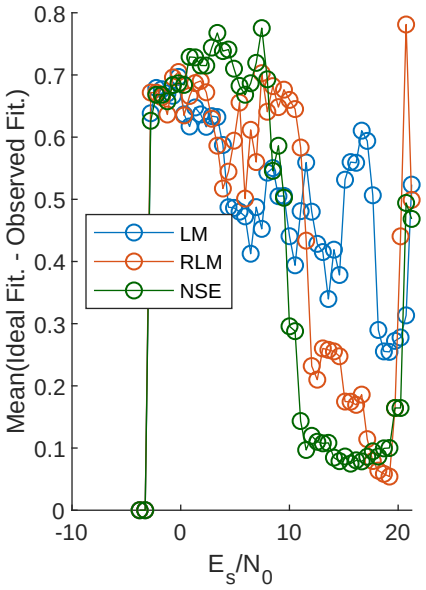


(a) Binned mean of fitness distance between ideal fitness and observed fitness.

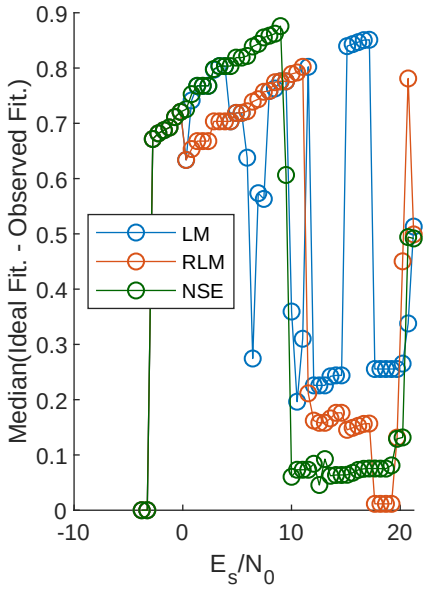


(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.68: Binned mean and binned median plots for the different CE training methods running the Power Saving mission on SNR profile 3. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

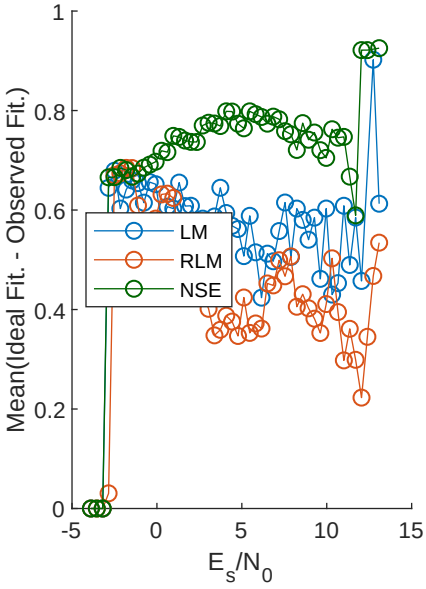


(a) Binned mean of fitness distance between ideal fitness and observed fitness.

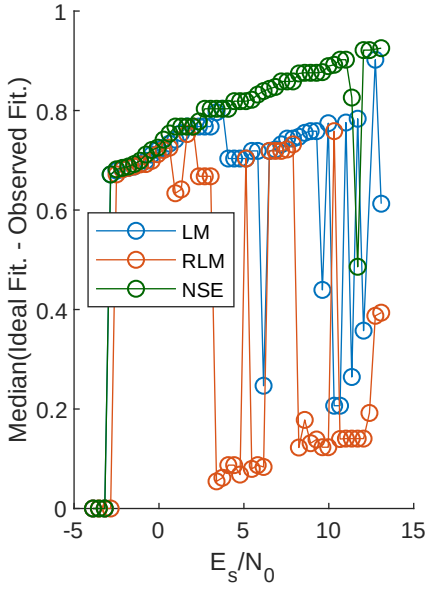


(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.69: Binned mean and binned median plots for the different CE training methods running the Power Saving mission on SNR profile 4. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.



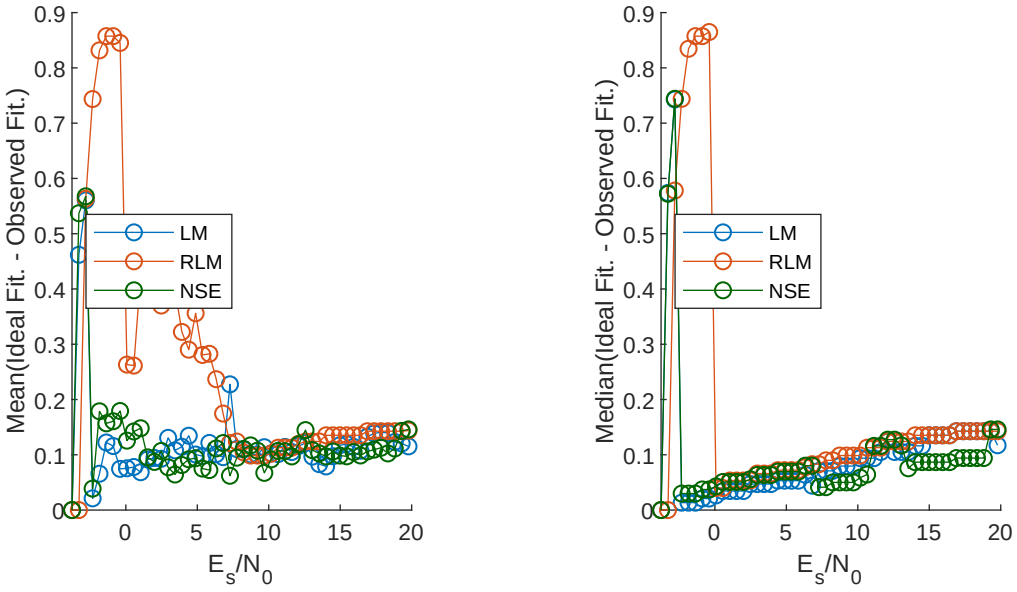
(a) Binned mean of fitness distance between ideal fitness and observed fitness.



(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.70: Binned mean and binned median plots for the different CE training methods running the Power Saving mission on SNR profile 5. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

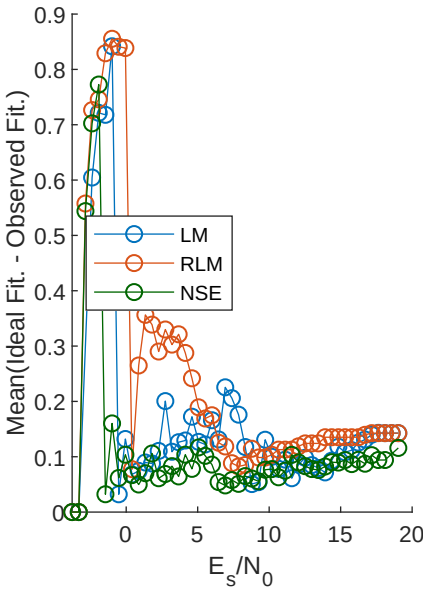
B.3.3 Emergency Mission



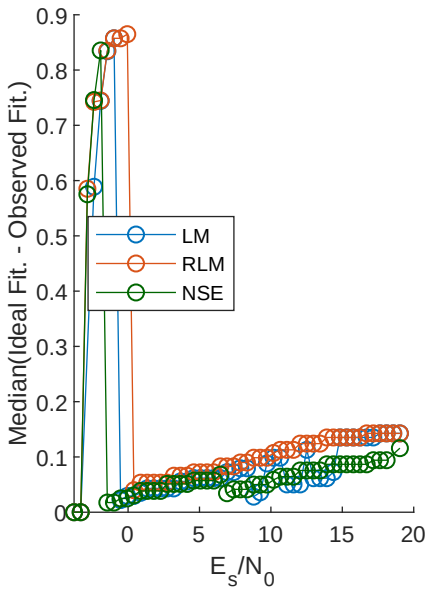
(a) Binned mean of fitness distance between ideal fitness and observed fitness.

(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.71: Binned mean and binned median plots for the different CE training methods running the Emergency mission on SNR profile 1. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

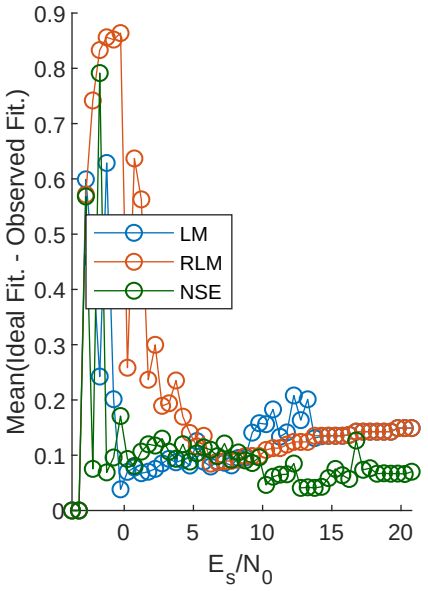


(a) Binned mean of fitness distance between ideal fitness and observed fitness.

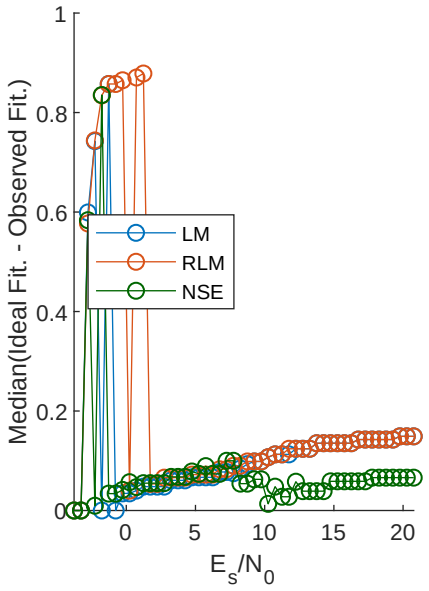


(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.72: Binned mean and binned median plots for the different CE training methods running the Emergency mission on SNR profile 2. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

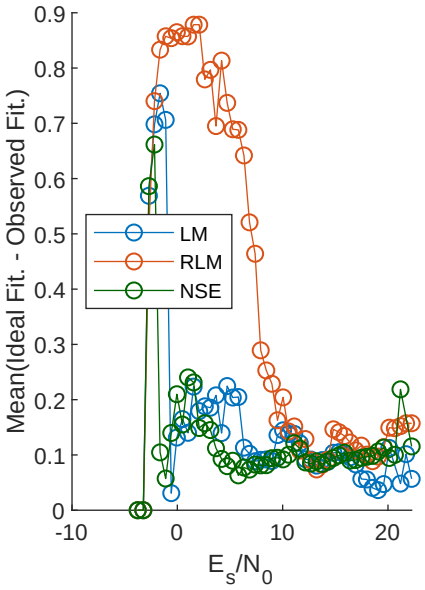


(a) Binned mean of fitness distance between ideal fitness and observed fitness.

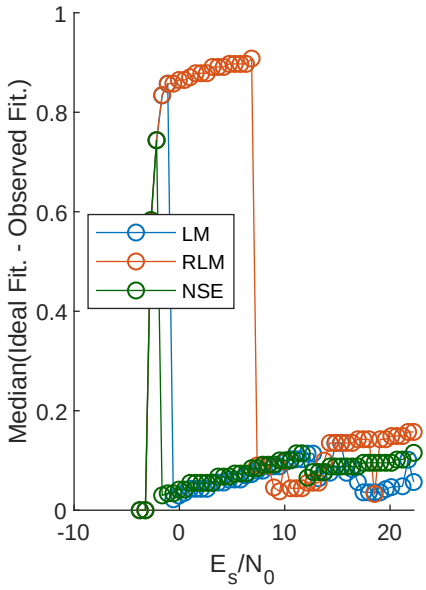


(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.73: Binned mean and binned median plots for the different CE training methods running the Emergency mission on SNR profile 3. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

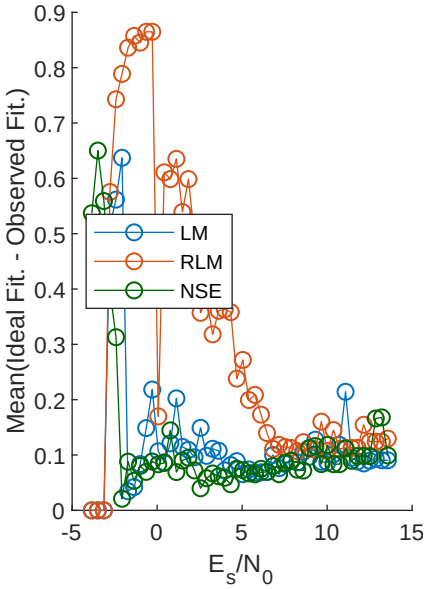


(a) Binned mean of fitness distance between ideal fitness and observed fitness.

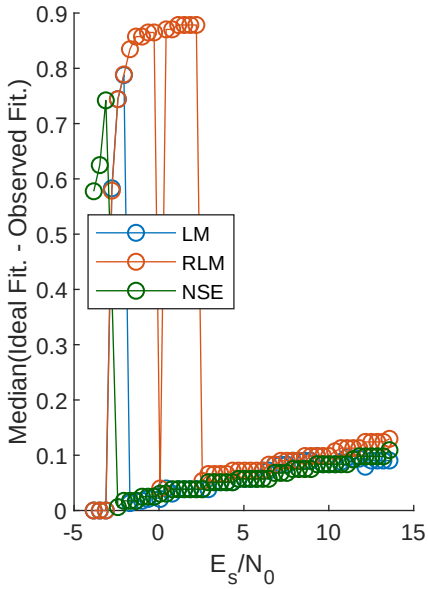


(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.74: Binned mean and binned median plots for the different CE training methods running the Emergency mission on SNR profile 4. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.



(a) Binned mean of fitness distance between ideal fitness and observed fitness.



(b) Binned median of fitness distance between ideal fitness and observed fitness.

Figure B.75: Binned mean and binned median plots for the different CE training methods running the Emergency mission on SNR profile 5. Both the mean and median are derived by binning the fitness scores by the E_s/N_0 values observed at the same time as the scores.

Appendix C

MATLAB Code

C.1 Preface

C.2 log_parser.m

```
clear all
close all
clc
COMPUTE_IDEAL_FITNESS = 1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Parse Log Files
fid = fopen('logging.txt','r');
A=textscan(fid,'%s','Delimiter','\n');
fclose(fid)

IterationA = strfind(A{1,1}, '::Iteration');
Iteration = find(not(cellfun('isempty', IterationA)));
B=zeros(length(Iteration),1);
for i=1:length(Iteration)
    B(i) = str2double(A{1,1}{Iteration(i),1}(14:end));
end
Iteration=[];
Iteration=B;

StartTimeA = strfind(A{1,1}, '::Start Time');
StartTime = find(not(cellfun('isempty', StartTimeA)));
B=cell(length(StartTime),1);
for i=1:length(StartTime)
    B{i} = A{1,1}{StartTime(i),1}(15:end);
end
StartTime=[];
StartTime=B;

ActionTypeA = strfind(A{1,1}, '::Action Type');
ActionType = find(not(cellfun('isempty', ActionTypeA)));
B=cell(length(ActionType),1);
for i=1:length(ActionType)
    B{i} = A{1,1}{ActionType(i),1}(16:end);
    if(strcmp(strtrim(B{i}),'Exploiting'))
        C(i) = 1;
    else
        C(i) = 0;
    end
end
ActionType=[];
ActionType=C;

ActionChosenTimeA = strfind(A{1,1}, '::Action Chosen');
ActionChosenTime = find(not(cellfun('isempty', ActionChosenTimeA)));
B=cell(length(ActionChosenTime),1);
for i=1:length(ActionChosenTime)
    B{i} = A{1,1}{ActionChosenTime(i),1}(18:end);
end
ActionChosenTime=[];
ActionChosenTime=B;

ActionIDA = strfind(A{1,1}, '::Action ID');
ActionID = find(not(cellfun('isempty', ActionIDA)));
B=zeros(length(ActionID),1);
```

```
for i=1:length(ActionID)
    B(i) = str2double(A{1,1}{ActionID(i),1}(14:end));
end
ActionID=[];
ActionID=B;

ActionParamsA = strfind(A{1,1}, '::Action Params');
ActionParams = find(not(cellfun('isempty', ActionParamsA)));
B=zeros(length(ActionParams),6);
for i=1:length(ActionParams)
    B(i,:) = str2num(A{1,1}{ActionParams(i),1}(18:end));
end
ActionParams=[];
ActionParams=B;

ActionSentTimeA = strfind(A{1,1}, '::Action Sent');
ActionSentTime = find(not(cellfun('isempty', ActionSentTimeA)));
B=cell(length(ActionSentTime),1);
for i=1:length(ActionSentTime)
    B{i} = A{1,1}{ActionSentTime(i),1}(16:end);
end
ActionSentTime=[];
ActionSentTime=B;

ActionRecdTimeA = strfind(A{1,1}, '::Action Received');
ActionRecdTime = find(not(cellfun('isempty', ActionRecdTimeA)));
B=cell(length(ActionRecdTime),1);
for i=1:length(ActionRecdTime)
    B{i} = A{1,1}{ActionRecdTime(i),1}(20:end);
end
ActionRecdTime=[];
ActionRecdTime=B;

RecvLockA = strfind(A{1,1}, '::Receive Lock');
RecvLock = find(not(cellfun('isempty', RecvLockA)));
B=zeros(length(RecvLock),1);
for i=1:length(RecvLock)
    B(i) = str2double(A{1,1}{RecvLock(i),1}(17:end));
end
RecvLock=[];
RecvLock=B;

MeasRecvdA = strfind(A{1,1}, '::Measurement Received');
MeasRecvd = find(not(cellfun('isempty', MeasRecvdA)));
B=zeros(length(MeasRecvd),6);
for i=1:length(MeasRecvd)
    B(i,:) = str2num(A{1,1}{MeasRecvd(i),1}(25:end));
end
MeasRecvd=[];
MeasRecvd=B;

ObjecFitA = strfind(A{1,1}, '::Objective Fitnesses Observed');
ObjecFit = find(not(cellfun('isempty', ObjecFitA)));
B=zeros(length(ObjecFit),6);
for i=1:length(ObjecFit)
    B(i,:) = str2num(A{1,1}{ObjecFit(i),1}(33:end));
end
ObjecFit=[];
ObjecFit=B;

FitObsA = strfind(A{1,1}, '::Fitness Observed');
FitObs = find(not(cellfun('isempty', FitObsA)));
B=zeros(length(FitObs),1);
for i=1:length(FitObs)
    B(i) = str2double(A{1,1}{FitObs(i),1}(21:end));
end
FitObs=[];
FitObs=B;

TrainingA = strfind(A{1,1}, '::Training');
Training = find(not(cellfun('isempty', TrainingA)));
B=cell(length(Training),1);
C=zeros(length(Training),1);
for i=1:length(Training)
    B{i} = A{1,1}{Training(i),1}(13:end);
    if(strcmp(B{i},'Yes'))
        C(i) = 1;
    else
        C(i) = 0;
    end
end
Training=[];
Training=C;

EndTimeA = strfind(A{1,1}, '::End Time');
EndTime = find(not(cellfun('isempty', EndTimeA)));
B=cell(length(EndTime),1);
for i=1:length(EndTime)
    B{i} = A{1,1}{EndTime(i),1}(13:end);
end
EndTime=[];
EndTime=B;
```

```
149
150 ModListA = strfind(A{1,1}, '::cogEngParams.nnAppSpec_modList:');
151 ModList = find(not(cellfun('isempty', ModListA)));
152 ModList = str2num(A{1,1}{ModList,1}(34:end));
153
154 CodListA = strfind(A{1,1}, '::cogEngParams.nnAppSpec_codList:');
155 CodList = find(not(cellfun('isempty', CodListA)));
156 CodList = str2num(A{1,1}{CodList,1}(34:end));
157
158 RollOffListA = strfind(A{1,1}, '::cogEngParams.nnAppSpec_rollOffList:');
159 RollOffList = find(not(cellfun('isempty', RollOffListA)));
160 RollOffList = str2num(A{1,1}{RollOffList,1}(38:end));
161
162 SymbolRateListA = strfind(A{1,1}, '::cogEngParams.nnAppSpec_symbolRateList:');
163 SymbolRateList = find(not(cellfun('isempty', SymbolRateListA)));
164 SymbolRateList = str2num(A{1,1}{SymbolRateList,1}(41:end));
165
166 TransmitPowerListA = strfind(A{1,1}, '::cogEngParams.nnAppSpec_transmitPowerList:');
167 TransmitPowerList = find(not(cellfun('isempty', TransmitPowerListA)));
168 TransmitPowerList = str2num(A{1,1}{TransmitPowerList,1}(44:end));
169
170 ModCodListA = strfind(A{1,1}, '::cogEngParams.nnAppSpec_modCodList:');
171 ModCodList = find(not(cellfun('isempty', ModCodListA)));
172 ModCodList = str2num(A{1,1}{ModCodList,1}(37:end));
173
174 ActionTableStartIdx = strfind(A{1,1}, '::Action List:');
175 ActionTableStartIdx = find(not(cellfun('isempty', ActionTableStartIdx)))+1;
176 ActionTableEndIdx = strfind(A{1,1}, '::Action Idxs:');
177 ActionTableEndIdx = find(not(cellfun('isempty', ActionTableEndIdx)))-3;
178 ActionTable = zeros(ActionTableEndIdx-ActionTableStartIdx+1,length(str2num(A{1,1}{
      ActionTableStartIdx,1}(1:end))));
179 for i=1:size(ActionTable,1)
180     ActionTable(i,:) = str2num(A{1,1}{ActionTableStartIdx+i-1,1});
181 end
182
183 WeightVectorA = strfind(A{1,1}, '::cogEngParams.cogeng_fitnessWeights:');
184 WeightVector = find(not(cellfun('isempty', WeightVectorA)));
185 WeightVector = str2num(A{1,1}{WeightVector,1}(38:end));
186
187 % Open NewTec Log
188 bypassLogName = dir('bypass_log*');
189 fid = fopen(bypassLogName.name,'r');
190 A=textscan(fid,'%s','Delimiter','\n');
191 fclose(fid)
192
193 FrameTimeA = strfind(A{1,1}, 'TIMESTAMP=');
194 FrameTime = find(not(cellfun('isempty', FrameTimeA)));
195 B=cell(length(FrameTime),1);
196 for i=1:length(FrameTime)
197     B{i} = A{1,1}{FrameTime(i),1}((FrameTimeA{FrameTime(i),1}-1)+11:(FrameTimeA{FrameTime(i)
      ,1}-1)+22);
198 end
199 FrameTime=[];
200 FrameTime=B;
201
202 % Open FER Curves
203 fid = fopen('ferCurves.txt','r');
204 A=textscan(fid,'%s','Delimiter','\n');
205 fclose(fid)
206
207 FERCurves = cell(size(A,1),2);
208 for i=1:size(A{1,1},1)
209     t = str2num(A{1,1}{i,1});
210     FERCurves(i,1) = {t(1:2:end)};
211     FERCurves(i,2) = {t(2:2:end)};
212 end
213
214 %%%%
```

```
237     end
238 end
239
240 % Convert date vectors into seconds. (doesn't work if spans multiple days)
241 StartTimeSec = zeros(length(StartTime),1);
242 ActionChosenTimeSec = zeros(length(StartTime),1);
243 ActionSentTimeSec = zeros(length(StartTime),1);
244 ActionRecdTimeSec = zeros(length(StartTime),1);
245 EndTimeSec = zeros(length(StartTime),1);
246 for i=1:length(StartTime)
247     StartTimeSec(i) = (datenum(StartTime{i,1}(13:end)) - floor(datenum(StartTime{i,1}(13:end))))
       *24*3600;
248     ActionChosenTimeSec(i) = (datenum(ActionChosenTime{i,1}(13:end)) - floor(datenum(
       ActionChosenTime{i,1}(13:end))))*24*3600;
249     ActionSentTimeSec(i) = (datenum(ActionSentTime{i,1}(13:end)) - floor(datenum(ActionSentTime{
       i,1}(13:end))))*24*3600;
250     ActionRecdTimeSec(i) = (datenum(ActionRecdTime{i,1}(13:end)) - floor(datenum(ActionRecdTime{
       i,1}(13:end))))*24*3600;
251     EndTimeSec(i) = (datenum(EndTime{i,1}(13:end)) - floor(datenum(EndTime{i,1}(13:end))))
       *24*3600;
252 end
253 FrameTimeSec = (datenum(FrameTime) - floor(datenum(FrameTime)))*24*3600;
254 INIT_TIME_SEC=StartTimeSec(1);
255
256 % use a sliding window of 40 ms and find when we received packets and when
257 % we didn't.
258 NewtecLck = zeros(length(ActionSentTimeSec),1);
259 for i=1:length(ActionSentTimeSec)
260     A = find((FrameTimeSec < ActionSentTimeSec(i)) & (FrameTimeSec > ActionSentTimeSec(i)-0.040)
       );
261     if(~isempty(A))
262         NewtecLck(i) = 1;
263     end
264 end
265
266 save('parsedLogsWorkspace.mat')
267
268
269 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
270 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
271 %% Plotting
272 % Plot SNR Profile, ViaSat Lock
273 figure
274 plot(ActionRecdTimeSec-INIT_TIME_SEC,MeasRecvd(:,1),'b',ActionRecdTimeSec(RecvLock==0)-
       INIT_TIME_SEC,MeasRecvd(RecvLock==0,1),'*r',...
275     'MarkerSize',1)
276 ylabel('EsN0 Profile (dB)')
277 xlabel('Time (sec)')
278 legend('EsN0 Profile','Unreliable Measurement')
279 % Plot FitObserved
280 figure
281 plot(ActionRecdTimeSec-INIT_TIME_SEC,FitObs.*NewtecLck,'b*','MarkerSize',3)
282 ylabel('Fit Observed')
283 xlabel('Time (sec)')
284 % Plot Histogram of Fit Observed (Log)
285 figure
286 [cnts,cntrs] = hist(FitObs.*NewtecLck,50);
287 bar(cntrs,cnts)
288 set(gca,'YScale','log')
289 xlabel('Fit Observed')
290 ylabel('Counts (Log Scale)')
291 % Plot Histogram of Fit Observed (Linear)
292 figure
293 [cnts,cntrs] = hist(FitObs.*NewtecLck,50);
294 bar(cntrs,cnts)
295 xlabel('Fit Observed')
296 ylabel('Counts')
297 % Plot Receiver Lock Analysis
298 figure
299 % 1: Newtec locked.
300 % -1: ViaSat locked
301 % -0.5: ViaSat locked, Newtec not locked.
302 % 0.5: Newtec locked, ViaSat not locked.
303 X = FitObs.*NewtecLck - FitObs.*RecvLock;
304 plot(ActionRecdTimeSec(NewtecLck==1)-INIT_TIME_SEC, 1*NewtecLck(NewtecLck==1),'g*', ...
305      ActionRecdTimeSec(X>0) -INIT_TIME_SEC, 0.5*NewtecLck(X>0),  'b*', ...
306      ActionRecdTimeSec(X<0) -INIT_TIME_SEC, -0.5*RecvLock(X<0),  'r*', ...
307      ActionRecdTimeSec(RecvLock==1) -INIT_TIME_SEC, -1*RecvLock(RecvLock==1),'k*')
308 set(gca,'YTickLabel',[])
309 xlabel('Time (s)')
310 legend('Newtec Locked','Newtec Locked AND ViaSat NOT Locked', ...
311        'ViaSat Locked AND Newtec NOT Locked', 'ViaSat Locked','Location','east')
312 title('Analysis of Receiver Lock')
313 % Plot MODCOD, TX Power, Rolloff vs. Time
314 figure
315 subplot(3,1,1)
316 plot(ActionSentTimeSec-INIT_TIME_SEC,ModcodChosen, 'b')
317 ylabel('MODCOD')
318 xlabel('Time (sec)')
319 subplot(3,1,2)
320 plot(ActionSentTimeSec-INIT_TIME_SEC,ActionParams(:,2), 'g')
321 ylabel('TX Power (dB)')
```

```
xlabel('Time (sec)')
subplot(3,1,3)
plot(ActionSentTimeSec-INIT_TIME_SEC,ActionParams(:,4), 'r')
ylabel('Roll Off')
xlabel('Time (sec)')
% Plot Action Update Rate Histo
figure
Y=(1./(ActionSentTimeSec(2:end)-ActionSentTimeSec(1:end-1)));
[cnts1,cntrs1] = hist(Y(ActionType(2:end)==1),linspace(min(Y),max(Y),100));
[cnts0,cntrs0] = hist(Y(ActionType(2:end)==0),linspace(min(Y),max(Y),100));
bar(cntrs1,cnts1,1,'FaceColor','g')
hold on;
bar(cntrs0,cnts0,1,'FaceColor','r')
hold off;
xlabel('Update Rate Jitter (Hz)')
ylabel('Counts')
legend('Exploit','Explore','Location','northwest')
% Plot Action Update Rate Histo (log scale)
figure
Y=(1./(ActionSentTimeSec(2:end)-ActionSentTimeSec(1:end-1)));
[cnts1,cntrs1] = hist(Y(ActionType(2:end)==1),linspace(min(Y),max(Y),100));
[cnts0,cntrs0] = hist(Y(ActionType(2:end)==0),linspace(min(Y),max(Y),100));
bar(cntrs1,cnts1,1,'FaceColor','g')
hold on;
bar(cntrs0,cnts0,1,'FaceColor','r')
hold off;
set(gca,'YScale','log')
xlabel('Update Rate Jitter (Hz)')
ylabel('Counts (Log Scale)')
legend('Exploit','Explore','Location','northwest')
% Plot Action Update Time Series
figure
X=ActionSentTimeSec(2:end);
Y=(1./(ActionSentTimeSec(2:end)-ActionSentTimeSec(1:end-1)));
plot(X(ActionType(2:end)==1)-INIT_TIME_SEC,Y(ActionType(2:end)==1),'b*',...
     X(ActionType(2:end)==0)-INIT_TIME_SEC,Y(ActionType(2:end)==0),'r*',...
     'MarkerSize',3)
xlabel('Time (sec)')
ylabel('Update Rate (Hz)')
legend('Exploit','Explore','Location','east')

% MASTER PLOT (Time Series)
figure
subplot(6,1,1)
plot(ActionRecdTimeSec-INIT_TIME_SEC,MeasRecvd(:,1),'b',ActionRecdTimeSec(RecvLock==0)-
     INIT_TIME_SEC,MeasRecvd(RecvLock==0,1),'*r',...
     'MarkerSize',1)
ylabel('EsN0 Profile (dB)')
xlabel('Time (sec)')
legend('EsN0 Profile','Unreliable Measurement','Location','northeast')
subplot(6,1,2)
plot(ActionRecdTimeSec-INIT_TIME_SEC,FitObs.*NewtecLck,'k*','MarkerSize',3)
ylabel('Fit Observed')
xlabel('Time (sec)')
subplot(6,1,3)
plot(ActionSentTimeSec-INIT_TIME_SEC,ModcodChosen, 'b')
ylabel('MODCOD')
xlabel('Time (sec)')
subplot(6,1,4)
plot(ActionSentTimeSec-INIT_TIME_SEC,ActionParams(:,2), 'g')
ylabel('TX Power (dB)')
xlabel('Time (sec)')
subplot(6,1,5)
plot(ActionSentTimeSec-INIT_TIME_SEC,ActionParams(:,4), 'r')
ylabel('Roll Off')
xlabel('Time (sec)')
subplot(6,1,6)
X=ActionSentTimeSec(2:end);
Y=(1./(ActionSentTimeSec(2:end)-ActionSentTimeSec(1:end-1)));
plot(X(ActionType(2:end)==1)-INIT_TIME_SEC,Y(ActionType(2:end)==1),'b*',...
     X(ActionType(2:end)==0)-INIT_TIME_SEC,Y(ActionType(2:end)==0),'r*',...
     'MarkerSize',3)
xlabel('Time (sec)')
ylabel('Update Rate (Hz)')
legend('Exploit','Explore','Location','southeast')
saveas(gcf,'MasterTimeSeriesPlots.fig')

% Master Plot (Histograms and Lock Analysis)
figure
subplot(2,3,4)
[cnts,cntrs] = hist(FitObs.*NewtecLck,50);
bar(cntrs,cnts)
set(gca,'YScale','log')
xlabel('Fit Observed')
ylabel('Counts (Log Scale)')
subplot(2,3,1)
[cnts,cntrs] = hist(FitObs.*NewtecLck,50);
bar(cntrs,cnts)
xlabel('Fit Observed')
ylabel('Counts')
subplot(2,3,2)
Y=(1./(ActionSentTimeSec(2:end)-ActionSentTimeSec(1:end-1)));
```

```
[cnts1,cntrs1] = hist(Y(ActionType(2:end)==1),linspace(min(Y),max(Y),100));
[cnts0,cntrs0] = hist(Y(ActionType(2:end)==0),linspace(min(Y),max(Y),100));
bar(cntrs1,cnts1,1,'FaceColor','g')
hold on;
bar(cntrs0,cnts0,1,'FaceColor','r')
hold off;
xlabel('Update Rate Jitter (Hz)')
ylabel('Counts')
legend('Exploit','Explore','Location','northwest')
subplot(2,3,5)
Y=(1./(ActionSentTimeSec(2:end)-ActionSentTimeSec(1:end-1)));
[cnts1,cntrs1] = hist(Y(ActionType(2:end)==1),linspace(min(Y),max(Y),100));
[cnts0,cntrs0] = hist(Y(ActionType(2:end)==0),linspace(min(Y),max(Y),100));
bar(cntrs1,cnts1,1,'FaceColor','g')
hold on;
bar(cntrs0,cnts0,1,'FaceColor','r')
hold off;
set(gca,'YScale','log')
xlabel('Update Rate Jitter (Hz)')
ylabel('Counts (Log Scale)')
legend('Exploit','Explore','Location','northwest')
subplot(2,3,3)
% 1: Newtec locked.
% -1: ViaSat locked
% -0.5: ViaSat locked, Newtec not locked.
% 0.5: Newtec locked, ViaSat not locked.
X = FitObs.*NewtecLck - FitObs.*RecvLock;
plot(ActionRecdTimeSec(NewtecLck==1)-INIT_TIME_SEC, 1*NewtecLck(NewtecLck==1),'g*', ...
     ActionRecdTimeSec(X>0) -INIT_TIME_SEC, 0.5*NewtecLck(X>0),  'b*', ...
     ActionRecdTimeSec(X<0) -INIT_TIME_SEC, -0.5*RecvLock(X<0),  'r*', ...
     ActionRecdTimeSec(RecvLock==1) -INIT_TIME_SEC, -1*RecvLock(RecvLock==1),'k*')
set(gca,'YTickLabel',[])
xlabel('Time (s)')
legend('Newtec Locked','Newtec Locked AND ViaSat NOT Locked', ...
       'ViaSat Locked AND Newtec NOT Locked', 'ViaSat Locked','Location','east')
title('Analysis of Receiver Lock')
subplot(2,3,6)
plot(ActionRecdTimeSec(Training==1)-INIT_TIME_SEC,Training(Training==1),'*r', ...
     ActionRecdTimeSec(Training==0)-INIT_TIME_SEC,Training(Training==0),'*g')
xlabel('Time (sec)')
legend('Training','Not Training','Location','east')
set(gca,'YTickLabel',[])
saveas(gcf,'MasterHistogramPlots.fig')


% Plot FitObserved vs Ideal
if(COMPUTE_IDEAL_FITNESS)
    figure
    subplot(2,1,1)
    plot(ActionRecdTimeSec-INIT_TIME_SEC,MeasRecvd(:,1),'b',ActionRecdTimeSec(RecvLock==0)-
    INIT_TIME_SEC,MeasRecvd(RecvLock==0,1),'*r',...
        'MarkerSize',1)
    ylabel('EsN0 Profile (dB)')
    xlabel('Time (sec)')
    legend('EsN0 Profile','Unreliable Measurement','Location','northeast')
    subplot(2,1,2)
    plot(ActionRecdTimeSec-INIT_TIME_SEC,FitObs,'ro', ...
         ActionRecdTimeSec-INIT_TIME_SEC,FitObs.*NewtecLck,'k*', ...
         ActionRecdTimeSec-INIT_TIME_SEC,OptimalFitnessObserved,'g', ...
         'MarkerSize',3)
    ylabel('Fit Observed')
    xlabel('Time (sec)')
    legend('Fitness Observed','Fitness After Post Processing','Ideal Fitness','Location','east')
    saveas(gcf,'FitObservedVersusIdeal.fig')

end

% MASTER PLOT (Time Series)
figure
subplot(4,1,1)
plot(ActionRecdTimeSec-INIT_TIME_SEC,FitObs.*NewtecLck)
ylabel('Fitness Observed')
xlabel('Time (sec)')
subplot(4,1,2)
plot(ActionRecdTimeSec-INIT_TIME_SEC,ObjecFit(:,1).*NewtecLck,'b',ActionRecdTimeSec-
    INIT_TIME_SEC,ObjecFit(:,2).*NewtecLck,'r')
ylabel('SubFit Observed')
xlabel('Time (sec)')
legend('Throughput','FER','Location','northeast')
subplot(4,1,3)
plot(ActionRecdTimeSec-INIT_TIME_SEC,ObjecFit(:,3).*NewtecLck,'b',ActionRecdTimeSec-
    INIT_TIME_SEC,ObjecFit(:,4).*NewtecLck,'r')
ylabel('SubFit Observed')
xlabel('Time (sec)')
legend('Bandwidth','Spectral Efficiency','Location','northeast')
subplot(4,1,4)
plot(ActionRecdTimeSec-INIT_TIME_SEC,ObjecFit(:,5).*NewtecLck,'b',ActionRecdTimeSec-
    INIT_TIME_SEC,ObjecFit(:,6).*NewtecLck,'r')
ylabel('SubFit Observed')
xlabel('Time (sec)')
legend('TX Power Efficiency','Power Consumption','Location','northeast')
saveas(gcf,'FitObservedTimeSeriesPlots.fig')
```

```
501
502 % VIRTUAL EXPLORATION ANALYSIS
503 figure
504 Y=FitObs.*NewtecLck;
505 subplot(2,1,1)
506 plot(ActionRecdTimeSec(ActionType(1:end)==0)-INIT_TIME_SEC,Y(ActionType(1:end)==0),'r*',
        ActionRecdTimeSec-INIT_TIME_SEC,Y,'b')
507 xlabel('Time (s)')
508 ylabel('Fitness Observed')
509 legend('Exploration Fitness','Fitness Observed','Location','northeast')
510 subplot(2,1,2)
511 %[cnts1,cntrs1] = hist(Y(ActionType(1:end)==1),linspace(min(Y),max(Y),100));
512 [cnts0,cntrs0] = hist(Y(ActionType(1:end)==0),linspace(min(Y),max(Y),100));
513 %bar(cntrs1,cnts1,1,'FaceColor','g')
514 %hold on;
515 bar(cntrs0,cnts0,1,'FaceColor','r')
516 hold off;
517 set(gca,'YScale','log')
518 set(gca,'Xlim',[0 1])
519 xlabel('Fit Observed')
520 ylabel('Counts (Log Scale)')
521 %legend('Exploit','Explore','Location','northwest')
522 saveas(gcf,'VirtualExplorationAnalysis.fig')
```

C.3 findTheoreticalBestAction.m

```
function [actionID, fitObserv, fitParams] = findTheoreticalBestAction(actionTable, esN0,
    FERCurves, weightVector, ModList, CodList, ModCodList)

    maxBER = 1e-6;
  RsMax = max(actionTable(:,1));
  RsMin = min(actionTable(:,1));
  BWMin = RsMin*(1+min(actionTable(:,4)));
  BWMax = RsMax*(1+max(actionTable(:,4)));
  TMax = RsMax*log2(max(actionTable(:,5)))*max(actionTable(:,6));
  TMin = RsMin*log2(min(actionTable(:,5)))*min(actionTable(:,6));
  EsMinLin = 10.0^(min(actionTable(:,2))/10);
  EsMaxLin = 10.0^(max(actionTable(:,2))/10);
  PConsumMinLin = EsMinLin*RsMin;
  PConsumMaxLin = EsMaxLin*RsMax;
  SpectEffMin = log2(min(actionTable(:,5)))*min(actionTable(:,6))/(1+max(actionTable(:,4)));
  SpectEffMax = log2(max(actionTable(:,5)))*max(actionTable(:,6))/(1+min(actionTable(:,4)));
  PEffMaxLog10 = log10(log2(max(actionTable(:,5)))*max(actionTable(:,6))/(EsMinLin*RsMin));
  PEffMinLog10 = log10(log2(min(actionTable(:,5)))*min(actionTable(:,6))/(EsMaxLin*RsMax));
  berDBMax = -10*log10(maxBER);
  berDBMin = -10*log10(1);

    Rs = actionTable(:,1);
    esAdd = actionTable(:,2);
    roll_off = actionTable(:,4);
    M = actionTable(:,5);
    rate = actionTable(:,6);

    measuredPowConsumedLin = 10.0.^(esAdd/10.0).*Rs;
    measuredPowConsumedLinComplement = PConsumMaxLin+PConsumMinLin - measuredPowConsumedLin;
    measuredPowEfficiencyLog10 = log10((log2(M).*rate)./measuredPowConsumedLin);
    measuredBandwidth = Rs.*(1.0+roll_off);
    measuredThroughput = Rs.*log2(M).*rate;
    measuredSpectralEff = log2(M).*rate./(1.0+roll_off);
    measuredBEREst = estimateBER(esN0,M,rate,FERCurves,ModList, CodList, ModCodList);
    measuredBEREstdB = -10.0.*log10(measuredBEREst);

    %populate observed params, normalized to [0,1]
    fitObservedParams = zeros(size(actionTable,1),6);
    fitObservedParams(:,1) = (measuredThroughput-TMin)/(TMax-TMin);
    fitObservedParams(:,2) = (measuredBEREstdB-berDBMin)/(berDBMax-berDBMin);
    fitObservedParams(:,3) = (measuredBandwidth-BWMin)/(BWMax-BWMin);
    fitObservedParams(:,4) = (measuredSpectralEff-SpectEffMin)/(SpectEffMax-SpectEffMin);
    fitObservedParams(:,5) = (measuredPowEfficiencyLog10-PEffMinLog10)/(PEffMaxLog10-
     PEffMinLog10);
    fitObservedParams(:,6) = (measuredPowConsumedLinComplement-PConsumMinLin)/(PConsumMaxLin-
     PConsumMinLin);

    fitObserved = sum(fitObservedParams .* repmat(weightVector,size(actionTable,1),1),2);
    fitObserved = (fitObservedParams(:,2)~=0.0).*fitObserved; %reality check

    [fitObserv,actionID] = max(fitObserved);
    fitParams = fitObservedParams(actionID);

end

function FERlin = estimateBER(EsN0dB,M,rate,FERCurves,ModList,CodList,ModCodList)
    FERlog10 = zeros(size(M,1),1);
    FERlin = zeros(size(M,1),1);

  %find modcod
    Modcod = zeros(size(M,1),1);
    for i=1:size(M,1)
        Modcod(i) = ModCodList((abs(ModList-M(i))<0.0001) & ((abs(CodList-rate(i))<0.0001)));
    end

  %find closest bounding points on curve
  minPoint = zeros(size(Modcod,1),1);
  maxPoint = zeros(size(Modcod,1),1);

    for j=1:size(Modcod,1)
        modcodChosenIdx = find(ModCodList == Modcod(j));
        for i=1:size(FERCurves{modcodChosenIdx,1},2)
            if(EsN0dB>=FERCurves{modcodChosenIdx,1}(i))
                minPoint(j) = i;
            end
            if(EsN0dB < FERCurves{modcodChosenIdx,1}(i) && maxPoint(j)==0)
                maxPoint(j) = i;
            end
        end

        if(minPoint(j)==0) %to the left of the fer curve
            minPoint(j) = maxPoint(j); %use first two points and linearly interpolate leftward
            maxPoint(j) = maxPoint(j)+1;
        end
        if(maxPoint(j)==0)  %to the right of the fer cruve
            maxPoint(j) = minPoint(j); %use last two points and linearly interpolate rightward
            minPoint(j) = minPoint(j)-1;
        end
```

```
86
87          FERlog10(j) = ((log10(FERCurves{modcodChosenIdx,2}(maxPoint(j)))-log10(FERCurves{
        modcodChosenIdx,2}(minPoint(j))))/(FERCurves{modcodChosenIdx,1}(maxPoint(j))-FERCurves{
        modcodChosenIdx,1}(minPoint(j)))) ...
88                      *(EsN0dB-FERCurves{modcodChosenIdx,1}(minPoint(j)))+log10(FERCurves{
        modcodChosenIdx,2}(minPoint(j)));
89
90          %hard limit at 0 and -12
91          if(FERlog10(j)>=0)
92              FERlog10(j)=0;
93          end
94          if(FERlog10(j)<-6)
95              FERlog10(j) = -6;
96          end
97          %convert to linear
98          FERlin(j) = 10^FERlog10(j);
99
100     end
101
102
103 end
```

C.4 CE-LM MATLAB Simulation

```
rng('shuffle')
% clear all
close all; clc

for iterations=1:1
    iterations
    episode_dur=21600; %only for analysis
    mission_iter = 1;
    for mission_num=[6,5,3] %[1:5] Loop over different mission profiles/fitness functions
        clearvars -except mission_num episode_dur iterations numIterations f_observed_save f_observed2_save mission_iter
        f_observed=[];
        f_observed2=[];

        load('ber_functions_3') %loads DVB-S2 (Long frames) BER curve functions for online BER estimation using SNR measurements

        %% Build NN structures

        %----------------Baseline LM implentation----------------------
        %NN Explore (NN1)
        net=network(1,3,[0;0;0], [1 ; 0 ; 0], [ 0 0 0; 1 0 0; 0 1 0], [0 0 1]);

        %NN input size
        net.inputs{1}.size=1;
        %NN input range values
        net.inputs{1}.range = [-1 1; -1 1; -1 1; -1 1; -1 1; -1 1; -1 1];

        %NN train function
        net.trainFcn = 'trainlm';
        %NN dataset division function (training, validation, test)
        net.divideFcn='dividerand'; % 70%,15%,15% default

        % NN output functions (help nntransfer)

        % NN1 Layers
        net.layers{1}.size = 7;
        net.layers{1}.transferFcn = 'logsig';
        net.layers{2}.size = 50;
        net.layers{2}.transferFcn = 'logsig';
        net.layers{3}.size = 1;
        net.layers{3}.transferFcn = 'purelin';

        %Early stop conditions
        net.trainParam.max_fail=20;
        net.trainParam.min_grad=1e-12;

        %Number of parallel NN
        numNN=20;
        NN = cell(1,numNN);

        %Flags [do not change]
        NN_train=0; %checks if NN was trained and controls when to train it again
        NN_train_2=0;
        NN_train_exploit=0;

        %---------------------------------------
        %NN Exploit (NN2)

        net_exploit=network(1,2,[0;0], [1 ; 0], [ 0 0 ; 1 0], [0 1]);

        %NN input size
        net_exploit.inputs{1}.size=1;
        %NN input range values
        net_exploit.inputs{1}.range = [-1 1; -1 1; -1 1; -1 1; -1 1; -1 1; -1 1];

        %NN train function
        net_exploit.trainFcn = 'trainlm';
        %NN dataset division function (training, validation, test)
        net_exploit.divideFcn='dividerand'; % 70%,15%,15% default

        % NN output functions (help nntransfer)

        %NN2 Layers
        net_exploit.layers{1}.size = 20;
        net_exploit.layers{1}.transferFcn = 'logsig';
        net_exploit.layers{2}.size = 1;
        net_exploit.layers{2}.transferFcn = 'purelin';

        %Early stop conditions
        net_exploit.trainParam.max_fail=20;
        net_exploit.trainParam.min_grad=1e-12;
        tmp_newWeights = getwb(net_exploit);
        tmp_newWeights = sqrt(2/20) * randn(size(tmp_newWeights)); % 2/(20+1)
        net_exploit = setwb(net_exploit,tmp_newWeights);
        %Number of parallel NN

        numNN_exploit=10;
```

```
87          NN_exploit = cell(6,numNN_exploit);
88          %Flag
89          NN_train_exploit=0; %checks if NN was trained and controls when to train it again
90
91          max_f_observed=0;
92
93          %% RL iterations
94
95          for iii=1:1
96              % Load Channel --> 0=GEO; 1=LEO
97              cn=1;
98
99              if cn==1
100                 % Fixed - LEO Channel [CLEAR SKY or RAIN]
101                 load ('L_fs.mat') %(LEO time series) Clear sky SNR profile at fixed ground
    receiver
102                 TOTAL=(L_fs-max(L_fs))*-1;
103
104             else
105                 % Fixed - GEO Channel [CLEAR SKY or RAIN]
106                 TOTAL=6*ones(1,episode_dur); %GEO clear sky SNR profile >>> 1000 seconds of
    constant 9 dB SNR profile
107             end
108             %Upsample attenuation time series to 10Hz
109             for i=1:1:length(TOTAL)
110                 TOTAL2(10*i-9:10*i)=TOTAL(i);
111             end
112
113             %% Initializing variables
114
115             %Adaptation parameters
116
117             % IN CASE **ANY** PARAMETER CHANGE ITS RANGE [MIN, MAX] the
118             % NORMALIZATION 'ps' function MUST BE LEARNED again !!!
119
120             mod_list = [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 8, 8, 8, 8, 8, 8, 16, 16, 16, 16, 16,
    16, 32, 32, 32, 32, 32];
121             cod_list = [1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 8/9, 9/10, 3/5, 2/3, 3/4,
    5/6, 8/9, 9/10, 2/3, 3/4, 4/5, 5/6, 8/9, 9/10, 3/4, 4/5, 5/6, 8/9, 9/10];
122
123             M_=[4 8 16 32];
124             k_=log2(M_);
125
126             BW_max=5e6; %MHz --> Max single-frequency SCaN testbed transponder bandwidth
127             BW_min=0.5e6; %MHz
128
129             roll_off=[0.2 0.3 0.35]; %Squared-root raised-cosine filter roll-off factor
130
131             Rs_max=BW_max/(1+max(roll_off)); %Max Symbol rate such that BW_max value is not
    compromised
132             Rs_min=BW_min/(1+min(roll_off));
133
134             T_max= Rs_max * log2(max(mod_list)) * max(cod_list); %Throughput in bits/sec
135             T_min= Rs_min * log2(min(mod_list)) * min(cod_list);
136
137             frame_size_=64800;% in bits --->Long-frame DVB-S2
138
139             % Ranges of adaptable parameters for value scalling of NN inputs
140             modcod_=1:length(mod_list); %Mod + Cod
141             Rs_=Rs_min:0.1*1e6:Rs_max;     %Symbol rate range
142             Es_=0:10; %Additional Es/No [dB] to boost signal before channel;
143
144             %Values used for normalization of monitored parameters
145             Es_min_lin=10^(min(Es_)/10);
146             Es_max_lin=10^(max(Es_)/10);
147
148             %Consumed Power
149             P_consu_min_lin=(Es_min_lin*Rs_min); %linear
150             P_consu_max_lin=(Es_max_lin*Rs_max);
151
152             %Spectral efficiency
153             spect_eff_max=max(log2(mod_list))*max(cod_list)/(1+min(roll_off));
154             spect_eff_min=min(log2(mod_list))*min(cod_list)/(1+max(roll_off));
155
156             %Consumed power efficiency
157             pwr_eff_max=(max(log2(mod_list))*max(cod_list))/(Es_min_lin*Rs_min);
158             pwr_eff_min=(min(log2(mod_list))*min(cod_list))/(Es_max_lin*Rs_max);
159
160             %SNR
161             max_SNR=12.9263; % [dB] Maximum link margin achieved during clear sky conditions for
     current predicted orbit (obtained from link budget)
162             max_SNR_lin=10^(max_SNR/10);
163
164             %Designer paremeters: comms target performance BER value
165             BER_max=1e-12;% max BER
166             ber_dB_max=-10.*log10(BER_max);
167             ber_dB_min=-10.*log10(1);
168
169             BER_min=1e-6;% min BER
170
171             BW=1e6; % Constant bandwidth [Hz]
172
```

```
173             %   -->AI parameters<--
174
175             %RL parameters
176             tr=0;% RL state reward threshold
177             % [no (gamma) discount factor ->REMOVED TEMPORAL DIFFERENCE]
178             episilon(1)=1;
179
180             %U matrix - all parameter combinations between Rs_,Es_,modcod_,roll_off
181             U=(combvec(Rs_,Es_,modcod_,roll_off))';
182
183             %Add additional features for NN training
184             U=[U zeros(size(U,1),1) zeros(size(U,1),1)];
185             U(:,5)=mod_list(U(:,3));
186             U(:,6)=cod_list(U(:,3));
187             modcod_map=U(:,3);
188             U(:,3)=log2(U(:,5)); %replaces modcod mapping by mod_order
189
190             %Final U structure for NN training [Rs_, Es_, log2(M), roll_off, M, (n/k)]
191
192             %Get Actions normalization function (rows are features). Used to build normalized
        training set
193             [~,ps] = mapminmax(U');
194
195             %Scaled action matrix U_out, range [0, 1]
196             U_out=U;
197             for s=1:size(U,2)
198                 U_out(:,s)=(U_out(:,s)-min(U_out(:,s)))/(max(U_out(:,s))-min(U_out(:,s))); %
        range [0,1]
199             end
200
201             %List to classify predicted modcod (used at NN2 output)
202             a_=unique(U_out(:,3));
203             b_=unique(U_out(:,5));
204             x=0;
205             for i=1:4
206                 for j=1:4
207                     x=x+1;
208                     sum_(x)=a_(i)+b_(j); %[0 0.14 0.33 0.44 0.47 0.66 0.77 0.8 1 1 1.1 1.14 1.33
         1.44 1.66 2] Classification targets are: [0 .4762 1.0952 2]
209                 end
210             end
211
212             s0=1;% Initialization control to start as random exploration until NN gets trained
213
214             %Communication mission phases/scenarios
215             %     [Thrp, BER, BW, Spec_eff, Pwr_eff, Pwr ];
216             mission =...
217                 [0.2  0.4  0.1   0.1   0.1   0.1;     %Launch/re-entry (1)
218                  0.5  0.3  0.05  0.05  0.05  0.05;    %Multimedia      (2)
219                  0.05 0.05 0.05  0.05  0.3   0.5;     %Power saving    (3)
220                  1/6  1/6  1/6   1/6   1/6   1/6;     %Normal          (4)
221                  0.05 0.05 0.4   0.4   0.05  0.05;    %Cooperation     (5)
222                  0.1  0.8  0.025 0.025 0.025 0.025]; %Emergency       (6)
223
224             w=mission(mission_num,:); %Select communication mission.
225
226             n=2; %Initial exploration probability divider
227
228             %NO Q-table anymore! :)
229
230             %%  MAIN SIMULATION ITERATION
231
232
233             episilon_reset_lim=4e-3; %reset episilon when episilon < episilon_reset_lim
234
235             %Log observables (for performance analysis only)
236             log_f_max_T=zeros(1,episode_dur);
237             log_f_min_BER=zeros(1,episode_dur);
238             log_f_min_P=zeros(1,episode_dur);
239             log_f_const_W=zeros(1,episode_dur);
240
241             %Normalizing all NN exploration inputs (actions)
242             input_explore_2 = mapminmax('apply',U',ps); %Applying normalization function ps to
        new training input
243
244             %Percentage to exclude from training buffer for next retraining
245             nn_delete=1;%1/4; %after training, delete 50% of oldest actions and retrain only
        after these percentage of training data is replaced by new training data
246
247             history_size=200; % <<<<< NN training window with UNIQUE actions (and its respective
         performance) >>> designer parameter ANALYZE IMPACT ON PERFORMANCE!!!!!
248             % Percentage of training data used for training [for online learning 100% history
        data can be used] (remaining training data is used for parellel testing)
249             nn_parallel_train=1; %(90% for training and 10% for parellel testing)
250
251             hist_count=0;
252             ind_mean=0;
253
254             temp_hist_=zeros(history_size,10); %Sliding window of last action-performance data [
        acdionID f_observed time_stamp]
255             hist_=zeros(history_size,size(U,2)+1); %[action_parameters f_observed]
256
```

```
257             action_pred=zeros(1,episode_dur);
258             action_discrete=zeros(1,episode_dur);
259 
260             %NN1 threshold prediction values for virtual exploration
261             perc_max_explore=.90; % threshold percentage of current achievable maximum predicted
     performance
262             rejection_rate=.95; %rejection percentage (rate) of exploring actions which
    performance predictions fall below the performance_threshold value
263 
264             train_ind=[];% indices when NN training took place
265 
266             track_exploit_err=0;
267 
268             explore_control=0; %used in while loop during explore_mode=1 ONLY (starts as 0
    always to be run once)
269 
270             first_explore=0;
271 
272             e_p=[]; %exploit performance (holds last exploitation performance value; if new/
    current exploitation action resuls in poorer performance than previous, roll-back last
    exploitation action
273             last_e_a=[-1 -1 -1 -1 -1 -1];%NaN NaN NaN NaN NaN NaN];
274 
275             elapsed_time=0; %time counter [seconds]
276             jji=0; %main discrete time index/packet counter
277             frame_dur=[];
278 
279             ind3=0;
280 
281             %% Main processing loop basedd on time duration (tracked by elapsed time, computing
    packet duration)
282             while elapsed_time<512%length(time_series(1).esno_viasat)%50%512
283                 jji=jji+1; %increments packet counter
284 
285                 % AI functions
286                 % [For implementation include parameter change monitoring feature]
287                 % If [Rs_, Es_, log2(M), roll_off, M, (n/k)] (either max, min, or any value
    within its ranges) changes, do the following:
288                 %   -Updates matrix U in case adaptable parameters changed
289                 %   -Update reference parameters for scaling
290 
291                 %Compute epsilon(jji) %e-greedy function of time
292                 %Decrease exploration rate/increase exploitation
293                 if jji>=history_size + 1 && epsilon(jji-1)> epsilon_reset_lim
294                     epsilon(jji)=1/n; % n is # os times system exploited config. After epsilon
    (jji-1)> epsilon_reset_lim, reset exploration probability.
295                     n=n+1;
296                 else
297                     epsilon(jji)=1;
298                     n=1;%reset epsilon
299                 end
300 
301 
302                 %-->> (Exploit or Explore) epsilon=Exploration probability <<--%
303                 if s0==0
304                     e_prob=rand; %Randomly (uniformly) pick Explore or Exploit
305 
306                     % [Explore]
307                     if e_prob<=epsilon(jji)
308                         expl=0; %flag
309 
310                         %-----NN1 Prediction--->Exploration
311                         %Predictions are done every iteration ONLY after NN is trained and ONLY
    if exploring
312                         if NN_train==1
313                             %Inputs
314                             input_explore=[input_explore_2; (ones(1,length(U)))*(((
    measured_SNR_lin/max_SNR_lin)-0.5)./0.5)]; %scaling to range [-1, 1]
315 
316                             %Predictions
317                             parfor i_NN=1:numNN
318                                 y_pred(i_NN,:)=NN{i_NN}(input_explore); %Explore NN prediction
319                             end
320                             f_predic=mean(y_pred); %Ensamble average prediction
321 
322                             %Logic to decide on actions predicted by NN
323                             perf_th=max(f_predic)*perc_max_explore; %current performance
    threshold
324 
325                             %get indexes of f_predic >= perf_th
326                             bigger_f_predic=find(f_predic>=perf_th);
327                             %get indexes of f_predic < perf_th
328                             smaller_f_predic=find(f_predic<perf_th);
329 
330                             NN_train_2=0;
331 
332                             if first_explore==1%right after training the action with max
    predicted performance is explored first (could guide Exploitation NN input)
333                                 ii=find(f_predic==max(f_predic));
334                                 ii=datasample(ii,1);
335                                 first_explore=0;
336                             else
```

```
337                                 if rand>=rejection_rate
338                                     % pick random action with performance < perf_th
339                                     if not(isempty(smaller_f_predic))
340                                         ii=datasample(smaller_f_predic,1);
341                                     else
342                                         ii=datasample(bigger_f_predic,1);
343                                     end
344                                 else
345                                     %pick random action with performance > perf_th
346                                     if not(isempty(bigger_f_predic))
347                                         ii=datasample(bigger_f_predic,1);
348                                     else
349                                         ii=datasample(smaller_f_predic,1);
350                                     end
351                                 end
352                             end
353                         end
354                         track_exploit_err=0; %reset when explore
355
356
357                         % [Exploit]
358                     else
359                         expl=1; %flag
360
361                         %------NN2 Prediction--->Exploitation
362                         %Predictions are done every iteration ONLY after NN is trained and ONLY
    if exploring
363                         if NN_train_exploit==1 %Use NN predictions if NN has been already
    trained
364
365                             %Inputs
366                             input_norm=[input_norm2 (measured_SNR_lin/max_SNR_lin)]';
367                             input_norm=(input_norm-0.5)./0.5; %scaling to range [-1, 1]
368
369                             %Predictions
370                             for n_i=1:numNN_exploit
371                                 for n_j=1:6
372                                     norm_action_pred(n_i,n_j)=NN_exploit{n_j,n_i}(input_norm); %
    Exploit NN prediction
373                                 end
374                             end
375                             norm_action=mean(norm_action_pred);%Ensamble average prediction
376
377                             %Classify modcod
378                             [~,modcod_class]=min(abs(sum_(1,1:5:16)-(norm_action(3)+norm_action
    (5))));
379
380                             %Denormalize predicted Action
381                             for s=1:size(U,2)
382                                 action_pred(s,jji)=(norm_action(s)*(max(U(:,s))-min(U(:,s))))+
    min(U(:,s)); %only for simulation analysis
383
384                                 %Classify denormalized values into executable action parameters
    [Action structure [Rs_, Es_, log2(M), roll_off, M, (n/k)] for NN training]
385                                 %Switcting between vectors (edges)
386                                 if s==1
387                                     edges=Rs_;
388                                 elseif s==2
389                                     edges=Es_;
390                                 elseif s==3
391                                     edges=2:5; %log2(M)
392                                 elseif s==4
393                                     edges=roll_off;
394                                 elseif s==5
395                                     edges=M_;
396                                 elseif s==6 %classify encoding rate based on the modulation
    order classified previously
397                                     if (action_discrete(5,jji))==4
398                                         edges=[1/4 1/3 2/5 1/2 3/5 2/3 3/4 4/5 5/6 8/9 9/10];
399                                     elseif (action_discrete(5,jji))==8
400                                         edges=[3/5 2/3 3/4 5/6 8/9 9/10];
401                                     elseif (action_discrete(5,jji))==16
402                                         edges=[2/3 3/4 4/5 5/6 8/9 9/10];
403                                     elseif (action_discrete(5,jji))==32
404                                         edges=[3/4 4/5 5/6 8/9 9/10];
405                                     end
406                                 end
407
408                                 if s==3 || s==5
409                                     min_ind=modcod_class;
410                                 else
411                                     [~,min_ind]=min(abs(action_pred(s,jji)-edges)); %
    clasification using minimum distance
412                                 end
413                                 action_discrete(s,jji)=edges(min_ind); %Discretized action
    values (just to comply with action table based on hardware capabilities)
414                             end
415
416                             [~,ii]=ismember(action_discrete(:,jji)', U, 'rows'); %finds the
    action ID of normalized predicted action within U
417
418
```

```
419                         else %If NN failed to be trained
420                             % Exploit actions from history in descending order of performance
421                         end
422                     end
423 
424                     %only applies for s0=1 mode (while NN have not been trained yet)
425                 else
426                     expl=0; %flag
427                     %s0 gets reset after NN is trained
428                     ii=ceil(rand*length(U));%First time always explore randomly
429                 end
430 
431                 action(jji)=ii; %Chosen action id time-series; for analysis only
432 
433                 %Parameters for chosen action --> %U structure [Rs_, P_, log2(M), roll_off, M, (
    n/k)] for NN training
434                 Rs=U(ii,1);
435                 Es_add=U(ii,2);
436                 k=U(ii,3);
437                 r_off=U(ii,4);
438                 M=U(ii,5);
439                 rate=U(ii,6);
440 
441                 %Action time series
442                 act_modcod_map(jji)=modcod_map(ii);
443                 act_Rs(jji)=U(ii,1);
444                 act_Es_add(jji)=U(ii,2);
445                 act_k(jji)=U(ii,3);
446                 act_r_off(jji)=r_off;
447                 act_M(jji)=U(ii,5);
448                 act_rate(jji)=U(ii,6);
449 
450 
451                 frame_dur(jji)=frame_size_/Rs; %Frame duration in secs
452                 elapsed_time=sum(frame_dur);
453 
454                 if ceil(elapsed_time/(512/5120))> length(TOTAL2) %for analysis only (stops
    script if there is no more data on synthetic snr time-series)
455                     break
456                 end
457 
458 
459                 %% Measured at Tx (sent to Rx as telemetry data):
460 
461                 %BEFORE BEING AFFECTED BY CHANNEL DYNAMICS
462 
463                 %Transmitted power
464                 measured_Es(jji)=Es_add;
465                 %Amplifier - Increase Es (energy per symbol)
466                 measured_SNR(jji)=TOTAL2(ceil(elapsed_time/(512/5120)));
467                 measured_SNR_lin=10^(measured_SNR(jji)/10);
468                 measured_SNR_lin_norm(jji)=measured_SNR_lin/max_SNR_lin;
469                 EsNo=measured_SNR(jji)+Es_add;
470 
471                 %Consumed additional power [dB]
472                 measured_P_consu(jji)=10*log10((10^(Es_add/10))*Rs);
473                 measured_P_consu_lin=10^(measured_P_consu(jji)/10);
474 
475                 %Power efficiency Rb/P_consu  [bits/sec/Watts]
476                 measured_Pwr_eff(jji)=(k*rate)/((10^(Es_add/10))*Rs);
477 
478                 %Bandwidth
479                 measured_W(jji)=Rs*(1+r_off);% [Hz]
480 
481                 %Throughput
482                 measured_T(jji)=Rs*k*rate;% [bits/sec]
483 
484 
485 
486                 %% AI at Rx:
487                 %Measured at Rx (AFTER BEING AFFECTED BY CHANNEL DYNAMICS):
488 
489                 %  This study uses SNR profiles. A real-world experiment is required to evaluate
     its performance while using real SNR measurements.
490 
491                 modcod_n=modcod_map(ii); %retrieves modcod ID that maps into BER_curve function
492                 eval(sprintf('ber_func = modcod_%d;', modcod_n)) %retrieves the proper BER_curve
     function
493                 if ber_func(EsNo)<0
494                     measured_BER_est(jji)=1e-12; %assigns very low value (non-zero)
495                 elseif ber_func(EsNo)==0
496                     measured_BER_est(jji)=1e-12; %assigns very low value (non-zero)
497                 elseif ber_func(EsNo)>1
498                     measured_BER_est(jji)=1; %holds value at max = 1
499                 else
500                     measured_BER_est(jji)=ber_func(EsNo); %assigns BER value as predicted by
    function
501                 end
502 
503                 %Spectral efficiency
504                 measured_spec_eff(jji)=k*rate/(1+r_off);
505 
```

```
506                 %(The following are sent out by the Tx as telemetry data:)
507                 %     -Transmitted power computed at Tx + additional power used above/below link budget estimated for clear sky operations
508                 %     -Bandwidth computed at Tx and not affected by channel (Assumed no presence of interferes).
509                 %     -Roll-off factor
510
511                 %Multi-objective fitness function (f_observed reference parameters)
512
513                 % Throughput
514                 f_max_T=measured_T(jji)/T_max;
515
516                 %BER
517                 f_min_BER_true=(100-(-1*(log10(BER_min/measured_BER_est(jji))))*(-100/(log10(BER_min))))/100; %Function value range from 1 to BER_min scaled to 0 to 1.
518                 if f_min_BER_true>=1
519                     f_min_BER=1;
520                 else
521                     f_min_BER=f_min_BER_true;
522                 end
523
524                 % Additional power
525                 f_min_P=Es_min_lin/(10^(Es_add/10));
526
527                 %Bandwidth
528                 if measured_W(jji)<=BW %Bandwidth
529                     f_const_W=1; %No penalty if W is smaller than target BW (cannot cause interference)
530                 else
531                     f_const_W=1-((measured_W(jji)-BW)/BW); %If W is more than double the target BW
532                     if f_const_W<0
533                         f_const_W=0;
534                     end
535                 end
536
537                 %Spectral efficiency
538                 spec_eff_max=log2((mod_list(end)))*(cod_list(end))/(1+min(roll_off));
539                 f_spec_eff=measured_spec_eff(jji)/spec_eff_max;
540
541
542                 %Observed state: fitness function
543                 f_observed2(jji,:) = [((measured_T(jji)-T_min)./(T_max-T_min)) ((-10.*log10(measured_BER_est(jji))-ber_dB_min)./(ber_dB_max-ber_dB_min)) ((measured_W(jji)-BW_min)./(BW_max-BW_min)) ((measured_spec_eff(jji)-spect_eff_min)./(spect_eff_max-spect_eff_min)) ((log10(measured_Pwr_eff(jji))-log10(pwr_eff_min))./(log10(pwr_eff_max)-log10(pwr_eff_min))) 1-((measured_P_consu_lin-P_consu_min_lin)./(P_consu_max_lin-P_consu_min_lin)) (measured_SNR_lin./max_SNR_lin)]; %range [0,1]
544                 f_observed(jji) = f_observed2(jji,1:end-1)*w'; %range [0,1] (SNR is not part of optimization goal)
545
546                 %Adapt/updates NN2 input whenever exploration finds a better performance
547                 if  s0==1 && hist_count==0
548                     e_p=f_observed(jji);
549                 elseif s0==1 && hist_count>0
550                     if f_observed(jji)>e_p
551                         e_p=f_observed(jji);
552                     end
553                 end
554
555                 if f_observed(jji)>max_f_observed
556                     max_f_observed=f_observed(jji); %global best known performance so far
557                     if expl==0
558                         input_norm2=f_observed2(jji,1:end-1);
559                     end
560                 else
561                     if expl==1 % [Exploit]
562                         if f_observed(jji)<e_p
563                             if e_p-f_observed(jji)>0.1 && (sum(input_norm2==last_e_a)==length(input_norm2)) %RESET "More efficient Recover Mode". Threshold value is a designer parameter (0.5 for specific missions, 0.1 for general)
564                                 s0=1; %enters exploration mode
565                                 %reset NN history
566                                 hist_count=0;
567                                 temp_hist_=zeros(history_size,10);
568                                 jji_reset(jji)=jji;
569                                 max_f_observed=0;
570                             elseif f_observed(jji)<e_p*0.9 %Quick "Recover Mode" using performances from the buffer. Triggers when 90% below previous exploration level
571                                 hist2=sortrows(temp_hist_,2);
572                                 ind3=ind3+1;
573                                 if ind3==history_size
574                                     ind3=1;
575                                 end
576                                 nn2=hist2(end-ind3,4:9);
577                                 input_norm2=[(nn2(:,1)-T_min)./(T_max-T_min) (-10.*log10(nn2(:,2))-ber_dB_min)./(ber_dB_max-ber_dB_min) (nn2(:,3)-BW_min)./(BW_max-BW_min) (nn2(:,4)-spect_eff_min)./(spect_eff_max-spect_eff_min) (log10(nn2(:,5))-log10(pwr_eff_min))./(log10(pwr_eff_max)-log10(pwr_eff_min)) 1-((nn2(:,6)-P_consu_min_lin)./(P_consu_max_lin-P_consu_min_lin))];
578                             elseif f_observed(jji)>e_p*0.9 && ind3>0 %Accepts new exploitation performance 90% above last exploitation threshold
```

```
                                e_p=f_observed(jji);
                                last_e_a=input_norm2;
                            else
                                input_norm2=last_e_a; %if exploiting and current exploitation performance is worse than previous exploitation performance; roll-back NN2 input
                            end
                        else
                            e_p=f_observed(jji); %tracks last exploitation performance
                            last_e_a=input_norm2; %tracks last exploitation NN2 input
                        end
                    end
                end

                %Logging function measurebles; for analysis only
                log_f_max_T(jji)=f_max_T;
                log_f_min_BER(jji)=f_min_BER;
                log_f_min_P(jji)=f_min_P;
                log_f_const_W(jji)=f_const_W;


                %-->> History sliding window (shared among Explore and Exploit NN's) <<--
                if isempty(find(temp_hist_(:,1)==ii)) %chosen action not present in sliding window
                    hist_count=hist_count+1; %populate
                    if hist_count<=history_size %if sliding window not full yet
                        ind_update=(temp_hist_(:,3)>=1);%index for update
                        temp_hist_(hist_count,:)= [ii f_observed(jji) 1 measured_T(jji) measured_BER_est(jji) measured_W(jji) measured_spec_eff(jji) measured_Pwr_eff(jji) measured_P_consu_lin measured_SNR_lin]; % [actions f_observed time_stamp]
                        temp_hist_(ind_update,3)=temp_hist_(ind_update,3)+1; %update all action IDs
                    else %if sliding window already full >>> replace
                        [~,jj]=max(temp_hist_(:,3)); %find action with oldest/highest time_stamp
                        temp_hist_(jj,:)= [ii f_observed(jji) 0 measured_T(jji) measured_BER_est(jji) measured_W(jji) measured_spec_eff(jji) measured_Pwr_eff(jji) measured_P_consu_lin measured_SNR_lin]; %replace it
                        temp_hist_(:,3)=temp_hist_(:,3)+1; %update all action IDs
                    end
                else %chosen action is already on sliding window, update it with most recent performance
                    ind_mean=find(temp_hist_(:,1)==ii);
                    ind_update=(temp_hist_(:,3)>0);%index for update
                    temp_hist_(ind_mean,:)= [ii f_observed(jji) 1 measured_T(jji) measured_BER_est(jji) measured_W(jji) measured_spec_eff(jji) measured_Pwr_eff(jji) measured_P_consu_lin measured_SNR_lin];
                    temp_hist_(ind_update,3)=temp_hist_(ind_update,3)+1; %update all action IDs
                end

                % ----> Train NN <----
                if hist_count==history_size % enables training after sliding window is full
                    train_ind=[train_ind jji]; % indices when NN training took place

                    %--------------Examples NN_explore--------------
                    % Populate with acitons (only after training window was built)
                    hist_(:,1:size(U,2))=U((temp_hist_(:,1)),:);
                    % Populate f_observed for non-repeating actions
                    hist_(:,end)=temp_hist_(:,2);
                    %Applying normalization function ps to new training input
                    pnewn = mapminmax('apply',hist_(:,1:size(U,2))',ps);

                    %NN1 training inputs
                    examples_input= [pnewn'];
                    examples_input=[examples_input (((temp_hist_(:,end)/max_SNR_lin)-0.5)./0.5)]; %scaled to range [-1,1]

                    %NN1 training outputs
                    examples_target= [hist_(:,end)]; %range [0, 1]

                    %Spliting inputs/outputs dataset (not needed for online operations)
                    %Splits training dataset into 90% training and 10% parallel testing
                    Q1 = floor(history_size*nn_parallel_train);
                    Q2 = history_size-Q1; %not needed for online operations
                    ind = randperm(history_size);
                    ind1 = ind(1:Q1);
                    ind2 = ind(Q1+(1:Q2)); %not needed for online operations
                    %Training
                    x1 = examples_input(ind1,:)';
                    y1 = examples_target(ind1,:)';
                    %Parallel testing (%Guarantees all nets use same test dataset)
                    x2 = examples_input(ind2,:)'; %not needed for online operations
                    y2 = examples_target(ind2,:)';%not needed for online operations


                    %Training NN1
                    tic
                    %par
                    parfor n_i=1:numNN
                        [NN{n_i},trainRecord{n_i}]=train(net,x1,y1);
                    end
                    elapsedTime = toc;

                    y1_results = NN{1}(x1);
```

```
                    err1 = perform(NN{1},y1,y1_results);
                    fprintf('Time to train explore network is %f, error = %f or %f \n',
    elapsedTime,err1,trainRecord{1}.best_tperf);%trainRecord{1}.best_tperf);
                    %flags0e
                    NN_train=1;
                    NN_train_2=1;
                    first_explore=1;


                    %--------------Examples NN_exploit--------------
                    %NN1 training inputs
                    examples_in_exploit = [(temp_hist_(:,4)-T_min)./(T_max-T_min) (-10.*log10(
    temp_hist_(:,5))-ber_dB_min)./(ber_dB_max-ber_dB_min) (temp_hist_(:,6)-BW_min)./(BW_max-
    BW_min) (temp_hist_(:,7)-spect_eff_min)./(spect_eff_max-spect_eff_min) (log10(temp_hist_
    (:,8))-log10(pwr_eff_min))./(log10(pwr_eff_max)-log10(pwr_eff_min)) 1-((temp_hist_(:,9)-
    P_consu_min_lin)./(P_consu_max_lin-P_consu_min_lin)) temp_hist_(:,10)./max_SNR_lin]; %range
    [0,1]
                    examples_in_exploit=(examples_in_exploit-0.5)./0.5; %scaled to range [-1, 1]

                    %NN1 training outputs
                    examples_out_exploit = [U_out(temp_hist_(:,1),:)]; %range [0, 1]

                    %Spliting inputs/outputs dataset (not needed for online operations)
                    %Splits training dataset into 90% training and 10% parallel testing
                    Q1 = floor(history_size*nn_parallel_train);
                    Q2 = history_size-Q1; %not needed for online operations
                    ind = randperm(history_size);
                    ind1 = ind(1:Q1);
                    ind2 = ind(Q1+(1:Q2)); %not needed for online operations
                    %Training
                    x1_exploit = examples_in_exploit(ind1,:)'; %not needed for online operations
                    y1_exploit = examples_out_exploit(ind1,:)'; %not needed for online
    operations
                    %Training NN2
                    parfor n_i=1:numNN_exploit
                        for n_j=1:6
                            NN_exploit{n_j,n_i}=train(net_exploit,x1_exploit,y1_exploit(n_j,:));
     %training per feeature NN
                        end
                    end
                    NN_train_exploit=1;

                    %--------------Reset sliding window--------------
                    %Delete x percentage of oldest actions
                    temp_hist_=sortrows(temp_hist_,3);
                    temp_hist_((history_size-(nn_delete*history_size))+1:end,:)=0;
                    hist_count=(history_size-(nn_delete*history_size)); %resets hist_count
                    ind3=1;

                    if s0==1
                        s0=0;%reset s0; allows NN's to be used
                    end
                end

            end %End of Main iteration (while)
        end %End of multiple iterations for different channels
        f_observed_save{mission_iter} = f_observed;
        f_observed2_save{mission_iter} = f_observed2;
        mission_iter = mission_iter + 1;
    end %End of mission loop
end
```

C.5 CE-RLM MATLAB Simulation

```
rng('shuffle')
% clear all
close all; clc

numIterations = 1;
f_observed_save = cell(numIterations,1);
f_observed2_save = cell(numIterations,1);
for iterations=1:numIterations
    iterations
    episode_dur=21600; %only for analysis
    mission_iter = 1;
    for mission_num=[6,5,3] %[1:5] Loop over different mission profiles/fitness functions
        clearvars -except mission_num episode_dur iterations numIterations f_observed_save
    f_observed2_save  mission_iter
        f_observed=[];
        f_observed2=[];


        load('ber_functions_3') %loads DVB-S2 (Long frames) BER curve functions for online BER
    estimation using SNR measurements

        %% Build NN structures

        %-----------------------------------------
        %NN Explore (NN1)
        %Number of parallel NN
        numNN=20;
        NN = cell(1,numNN);
        nWeights = 449;
        updateBatch = 0;
        updateRecurse = 1;
        if updateRecurse == 1
            NN_recurseMatrix = cell(1,numNN);
            NN_update_batch_size=10;
            alpha_explore = 0.97;
            for i = 1:numNN
                NN{i}=network(1,3,[0;0;0], [1 ; 0 ; 0], [ 0 0 0; 1 0 0; 0 1 0], [0 0 1]);

                %NN input size
                 NN{i}.inputs{1}.size=1;
                %NN input range values
                NN{i}.inputs{1}.range = [-1 1; -1 1; -1 1; -1 1; -1 1; -1 1; -1 1];

                %NN train function
                 NN{i}.trainFcn = 'trainlm';
                %NN dataset division function (training, validation, test)
                 NN{i}.divideFcn='dividerand'; % 70%,15%,15% default

                % NN output functions (help nntransfer)

                % NN1 Layers
                NN{i}.layers{1}.size = 7;
                NN{i}.layers{1}.transferFcn = 'logsig';
                NN{i}.layers{1}.initFcn = 'initnw';
                NN{i}.layers{2}.size = 50;
                NN{i}.layers{2}.transferFcn = 'logsig';
                NN{i}.layers{2}.initFcn = 'initnw';
                NN{i}.layers{3}.size = 1;
                NN{i}.layers{3}.transferFcn = 'purelin';
                NN{i}.layers{3}.initFcn = 'initnw';

                NN{i} = init(NN{i});
                tmp_newWeights = getwb(NN{i});
                tmp_newWeights =  sqrt(2/15)  * randn(size(tmp_newWeights)); % 2 / (n_out + n_in
    )
                NN{i} = setwb(NN{i},tmp_newWeights);

                 NN_recurseMatrix{i} = RecurseMatrix(-1,0,nWeights,alpha_explore);
            end
        end



        %Flags [do not change]
        NN_train=0; %checks if NN was trained and controls when to train it again
        NN_train_2=0;
        NN_train_exploit=0;
        alpha_exploit = 1;%alpha_explore;
        %-----------------------------------------
        %NN Exploit (NN2)

        %Number of parallel NN
        numNN_exploit=10;
        NN_exploit = cell(6,numNN_exploit);
        NN_recurseMatrix_exploit = cell(6,numNN_exploit);
        nWeights_exploit = 160;
        for i = 1:numNN_exploit
            for j = 1:6
```

```
86                  NN_exploit{j,i}=network(1,2,[0;0], [1 ; 0], [ 0 0 ; 1 0], [0 1]);
87                  %NN input size
88                  NN_exploit{j,i}.inputs{1}.size=1;
89                  %NN input range values
90                  NN_exploit{j,i}.inputs{1}.range = [-1 1; -1 1; -1 1; -1 1; -1 1; -1 1; -1 1];
91
92                  %NN train function
93                  NN_exploit{j,i}.trainFcn = 'trainlm';
94                  %NN dataset division function (training, validation, test)
95                  NN_exploit{j,i}.divideFcn='dividerand'; % 70%,15%,15% default
96                  NN_exploit{j,i}.layers{1}.initFcn = 'initnw';
97                  NN_exploit{j,i}.layers{2}.initFcn = 'initnw';
98                  %NN2 Layers
99                  NN_exploit{j,i}.layers{1}.size = 20;
100                 NN_exploit{j,i}.layers{1}.transferFcn = 'logsig';
101                 NN_exploit{j,i}.layers{2}.size = 1;
102                 NN_exploit{j,i}.layers{2}.transferFcn = 'purelin';
103
104                 %Early stop conditions
105                 NN_exploit{j,i}.trainParam.max_fail=20;
106                 NN_exploit{j,i}.trainParam.min_grad=1e-12;
107                 NN_exploit{j,i} = init(NN_exploit{j,i});
108
109                 tmp_newWeights = getwb(NN_exploit{j,i});
110                 tmp_newWeights = sqrt(2/20) * randn(size(tmp_newWeights)); % 2/(20+1)
111                 NN_exploit{j,i} = setwb(NN_exploit{j,i},tmp_newWeights);
112
113                 NN_recurseMatrix_exploit{j,i} = RecurseMatrix(-1,0,nWeights_exploit,
    alpha_exploit);
114             end
115         end
116
117
118         %Flag
119
120         NN_train_exploit=0; %checks if NN was trained and controls when to train it again
121
122         max_f_observed=0;
123         cnt_debug = 1;
124         %% RL iterations
125
126         for iii=1:1
127             % Load Channel --> 0=GEO; 1=LEO
128             cn=1;
129
130             if cn==1
131                 % Fixed - LEO Channel [CLEAR SKY or RAIN]
132                 load ('L_fs.mat') %(LEO time series) Clear sky SNR profile at fixed ground
    receiver
133                 TOTAL=(L_fs-max(L_fs))*-1;
134             else
135                 % Fixed - GEO Channel [CLEAR SKY or RAIN]
136                 TOTAL=6*ones(1,episode_dur); %GEO clear sky SNR profile >>> 1000 seconds of
    constant 9 dB SNR profile
137             end
138             %Upsample attenuation time series to 10Hz
139             for i=1:1:length(TOTAL)
140                 TOTAL2(10*i-9:10*i)=TOTAL(i);
141             end
142
143             %% Initializing variables
144
145             %Adaptation parameters
146
147             % IN CASE **ANY** PARAMETER CHANGE ITS RANGE [MIN, MAX] the
148             % NORMALIZATION 'ps' function MUST BE LEARNED again !!!
149
150             mod_list = [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 8, 8, 8, 8, 8, 8, 16, 16, 16, 16, 16,
    16, 32, 32, 32, 32, 32];
151             cod_list = [1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 8/9, 9/10, 3/5, 2/3, 3/4,
    5/6, 8/9, 9/10, 2/3, 3/4, 4/5, 5/6, 8/9, 9/10, 3/4, 4/5, 5/6, 8/9, 9/10];
152
153             M_=[4 8 16 32];
154             k_=log2(M_);
155
156             BW_max=5e6; %MHz --> Max single-frequency SCaN testbed transponder bandwidth
157             BW_min=0.5e6; %MHz
158
159             roll_off=[0.2 0.3 0.35]; %Squared-root raised-cosine filter roll-off factor
160
161             Rs_max=BW_max/(1+max(roll_off)); %Max Symbol rate such that BW_max value is not
    compromised
162             Rs_min=BW_min/(1+min(roll_off));
163
164             T_max= Rs_max * log2(max(mod_list)) * max(cod_list); %Throughput in bits/sec
165             T_min= Rs_min * log2(min(mod_list)) * min(cod_list);
166
167             frame_size_=64800;% in bits --->Long-frame DVB-S2
168
169             % Ranges of adaptable parameters for value scalling of NN inputs
170             modcod_=1:length(mod_list); %Mod + Cod
171             Rs_=Rs_min:0.1*1e6:Rs_max;     %Symbol rate range
```

```
Es_=0:10; %Additional Es/No [dB] to boost signal before channel;

%Values used for normalization of monitored parameters
Es_min_lin=10^(min(Es_)/10);
Es_max_lin=10^(max(Es_)/10);

%Consumed Power
P_consu_min_lin=(Es_min_lin*Rs_min); %linear
P_consu_max_lin=(Es_max_lin*Rs_max);

%Spectral efficiency
spect_eff_max=max(log2(mod_list))*max(cod_list)/(1+min(roll_off));
spect_eff_min=min(log2(mod_list))*min(cod_list)/(1+max(roll_off));

%Consumed power efficiency
pwr_eff_max=(max(log2(mod_list))*max(cod_list))/(Es_min_lin*Rs_min);
pwr_eff_min=(min(log2(mod_list))*min(cod_list))/(Es_max_lin*Rs_max);

%SNR
max_SNR=12.9263; % [dB] Maximum link margin achieved during clear sky conditions for current predicted orbit (obtained from link budget)
max_SNR_lin=10^(max_SNR/10);

%Designer paremeters: comms target performance BER value
BER_max=1e-12;% max BER
ber_dB_max=-10.*log10(BER_max);
ber_dB_min=-10.*log10(1);

BER_min=1e-6;% min BER

BW=1e6; % Constant bandwidth [Hz]

%   -->AI parameters<--

%RL parameters
tr=0;% RL state reward threshold
% [no (gamma) discount factor ->REMOVED TEMPORAL DIFFERENCE]
episilon(1)=1;

%U matrix - all parameter combinations between Rs_,Es_,modcod_,roll_off
U=(combvec(Rs_,Es_,modcod_,roll_off))';

%Add additional features for NN training
U=[U zeros(size(U,1),1) zeros(size(U,1),1)];
U(:,5)=mod_list(U(:,3));
U(:,6)=cod_list(U(:,3));
modcod_map=U(:,3);
U(:,3)=log2(U(:,5)); %replaces modcod mapping by mod_order

%Final U structure for NN training [Rs_, Es_, log2(M), roll_off, M, (n/k)]

%Get Actions normalization function (rows are features). Used to build normalized training set
[~,ps] = mapminmax(U');

%Scaled action matrix U_out, range [0, 1]
U_out=U;
for s=1:size(U,2)
    U_out(:,s)=(U_out(:,s)-min(U_out(:,s)))/(max(U_out(:,s))-min(U_out(:,s))); % range [0,1]
end

%List to classify predicted modcod (used at NN2 output)
a_=unique(U_out(:,3));
b_=unique(U_out(:,5));
x=0;
for i=1:4
    for j=1:4
        x=x+1;
        sum_(x)=a_(i)+b_(j); %[0 0.14 0.33 0.44 0.47 0.66 0.77 0.8 1 1 1.1 1.14 1.33 1.44 1.66 2] Classification targets are: [0 .4762 1.0952 2]
    end
end

s0=1;% Initialization control to start as random exploration until NN gets trained

%Communication mission phases/scenarios
%     [Thrp, BER, BW, Spec_eff, Pwr_eff, Pwr ];
mission =...
    [0.2   0.4   0.1    0.1    0.1    0.1;     %Launch/re-entry (1)
     0.5   0.3   0.05   0.05   0.05   0.05;    %Multimedia      (2)
     0.05  0.05  0.05   0.05   0.3    0.5;     %Power saving    (3)
     1/6   1/6   1/6    1/6    1/6    1/6;     %Normal          (4)
     0.05  0.05  0.4    0.4    0.05   0.05;    %Cooperation     (5)
     0.1   0.8   0.025  0.025  0.025  0.025];  %Emergency       (6)

w=mission(mission_num,:); %Select communication mission.

n=2; %Initial exploration probability divider

%NO Q-table anymore! :)
```

```
260         %%  MAIN SIMULATION ITERATION
261
262
263         episilon_reset_lim=4e-3; %reset episilon when episilon < episilon_reset_lim
264
265         %Log observables (for performance analysis only)
266         log_f_max_T=zeros(1,episode_dur);
267         log_f_min_BER=zeros(1,episode_dur);
268         log_f_min_P=zeros(1,episode_dur);
269         log_f_const_W=zeros(1,episode_dur);
270
271         %Normalizing all NN exploration inputs (actions)
272         input_explore_2 = mapminmax('apply',U',ps); %Applying normalization function ps to
    new training input
273
274         %Percentage to exclude from training buffer for next retraining
275         nn_delete=1;%1/4; %after training, delete 50% of oldest actions and retrain only
    after these percentage of training data is replaced by new training data
276         history_size=200; % <<<<< NN training window with UNIQUE actions (and its respective
     performance) >>> designer parameter ANALYZE IMPACT ON PERFORMANCE!!!!!
277
278         % Percentage of training data used for training [for online learning 100% history
    data can be used] (remaining training data is used for parellel testing)
279         nn_parallel_train=0.9; %(90% for training and 10% for parellel testing)
280         hist_count=0;
281         ind_mean=0;
282
283         temp_hist_=zeros(history_size,10); %Sliding window of last action-parformance data [
    acdionID f_observed time_stamp]
284         hist_=zeros(history_size,size(U,2)+1); %[action_parameters f_observed]
285
286         action_pred=zeros(1,episode_dur);
287         action_discrete=zeros(1,episode_dur);
288
289         %NN1 threshold prediction values for virtual exploration
290         perc_max_explore=.90; % threshold percentage of current achievable maximum predicted
     performance
291         rejection_rate=.95; %rejection percentage (rate) of exploring actions which
    performance predictions fall below the performance_threshold value
292
293         train_ind=[];% indices when NN training took place
294
295         track_exploit_err=0;
296
297         explore_control=0; %used in while loop during explore_mode=1 ONLY (starts as 0
    always to be run once)
298
299         first_explore=0;
300
301         e_p=[]; %exploit performance (holds last exploitation performance value; if new/
    current exploitation action resuls in poorer performance than previous, roll-back last
    exploitation action
302         last_e_a=[-1 -1 -1 -1 -1 -1];%[NaN NaN NaN NaN NaN NaN];
303
304         elapsed_time=0; %time counter [seconds]
305         jji=0; %main discrete time index/packet counter
306         frame_dur=[];
307
308         ind3=0;
309
310         %% Main processing loop basedd on time duration (tracked by elapsed time, computing
    packet duration)
311         while elapsed_time<512%length(time_series(2).esno_viasat)%50%512
312             jji=jji+1; %increments packet counter
313
314             % AI functions
315             % [For implementation include parameter change monitoring feature]
316             % If [Rs_, Es_, log2(M), roll_off, M, (n/k)] (either max, min, or any value
    within its ranges) changes, do the following:
317             %   -Updates matrix U in case adaptable parameters changed
318             %   -Update reference parameters for scaling
319
320             %Compute epsilon(jji) %e-greedy function of time
321             %Decrease exploration rate/increase exploitation
322             if jji>=(history_size + 1) && episilon(jji-1)> episilon_reset_lim
323                 episilon(jji)=1/n; % n is # os times system exploited config. After episilon
    (jji-1)> episilon_reset_lim, reset exploration probability.
324                 n=n+1;
325             else
326                 episilon(jji)=1;
327                 n=1;%reset epsilon
328             end
329
330
331             %-->> (Exploit or Explore) epsilon=Exploration probability <<--%
332             if s0==0
333                 e_prob=rand; %Randomly (uniformly) pick Explore or Exploit
334
335                 % [Explore]
336                 if e_prob<=episilon(jji)
337                     expl=0; %flag
338                     fprintf('Explore \n');
```

```
339                         %-----NN1 Prediction--->Exploration
340                         %Predictions are done every iteration ONLY after NN is trained and ONLY
    if exploring
341                         if NN_train==1
342                             %Inputs
343                             input_explore=[input_explore_2; (ones(1,length(U)))*(((
    measured_SNR_lin/max_SNR_lin)-0.5)./0.5)]; %scaling to range [-1, 1]
344
345                             %%% USAGE OF NNs
346                             %Predictions
347
348                             parfor i_NN=1:numNN
349                                 y_pred(i_NN,:)=NN{i_NN}(input_explore); %Explore NN prediction
350                             end
351                             ind_applicable = find(err < 1);
352                             f_predic=mean(y_pred(ind_applicable,:)); %Ensamble average
    prediction
353
354                             %Logic to decide on actions predicted by NN
355                             perf_th=max(f_predic)*perc_max_explore; %current performance
    threshold
356
357                             %get indexes of f_predic >= perf_th
358                             bigger_f_predic=find(f_predic>=perf_th);
359                             %get indexes of f_predic < perf_th
360                             smaller_f_predic=find(f_predic<perf_th);
361
362                             NN_parfortrain_2=0;
363
364                             if first_explore==1%right after training the action with max
    predicted performance is explored first (could guide Exploitation NN input)
365                                 ii=find(f_predic==max(f_predic));
366                                 ii=datasample(ii,1);
367                                 first_explore=0;
368                             else
369                                 if rand>=rejection_rate
370                                     % pick random action with performance < perf_th
371                                     if not(isempty(smaller_f_predic))
372                                         ii=datasample(smaller_f_predic,1);
373                                     else
374                                         ii=datasample(bigger_f_predic,1);
375                                     end
376                                 else
377                                     %pick random action with performance > perf_th
378                                     if not(isempty(bigger_f_predic))
379                                         ii=datasample(bigger_f_predic,1);
380                                     else
381                                         ii=datasample(smaller_f_predic,1);
382                                     end
383                                 end
384                             end
385                         end
386                         track_exploit_err=0; %reset when explore
387
388
389                         % [Exploit]
390                     else
391                         expl=1; %flag
392
393                         %-----NN2 Prediction--->Exploitation
394                         %Predictions are done every iteration ONLY after NN is trained and ONLY
    if exploring
395                         if NN_train_exploit==1 %Use NN predictions if NN has been already
    trained
396
397                             %Inputs
398                             input_norm=[input_norm2 (measured_SNR_lin/max_SNR_lin)]';
399                             input_norm=(input_norm-0.5)./0.5; %scaling to range [-1, 1]
400
401                             %%% NN_exploit usage
402                             %Predictions
403                             for n_i=1:numNN_exploit
404                                 parfor n_j=1:6
405                                     norm_action_pred(n_i,n_j)=NN_exploit{n_j,n_i}(input_norm); %
    Exploit NN prediction
406                                 end
407                             end
408                             norm_action=mean(norm_action_pred);%Ensamble average prediction
409
410                             %Classify modcod
411                             [~,modcod_class]=min(abs(sum_(1,1:5:16)-(norm_action(3)+norm_action
    (5))));
412
413                             %Denormalize predicted Action
414                             for s=1:size(U,2)
415                                 action_pred(s,jji)=(norm_action(s)*(max(U(:,s))-min(U(:,s))))+
    min(U(:,s)); %only for simulation analysis
416
417                                 %Classify denormalized values into executable action parameters
    [Action structure [Rs_, Es_, log2(M), roll_off, M, (n/k)] for NN training]
418                                 %Switcting between vectors (edges)
419                                 if s==1
```

```
                                    edges=Rs_;
                                elseif s==2
                                    edges=Es_;
                                elseif s==3
                                    edges=2:5; %log2(M)
                                elseif s==4
                                    edges=roll_off;
                                elseif s==5
                                    edges=M_;
                                elseif s==6 %classify encoding rate based on the modulation order classified previously
                                    if (action_discrete(5,jji))==4
                                        edges=[1/4 1/3 2/5 1/2 3/5 2/3 3/4 4/5 5/6 8/9 9/10];
                                    elseif (action_discrete(5,jji))==8
                                        edges=[3/5 2/3 3/4 5/6 8/9 9/10];
                                    elseif (action_discrete(5,jji))==16
                                        edges=[2/3 3/4 4/5 5/6 8/9 9/10];
                                    elseif (action_discrete(5,jji))==32
                                        edges=[3/4 4/5 5/6 8/9 9/10];
                                    end
                                end

                                if s==3 || s==5
                                    min_ind=modcod_class;
                                else
                                    [~,min_ind]=min(abs(action_pred(s,jji)-edges)); %clasification using minimum distance
                                end
                                action_discrete(s,jji)=edges(min_ind); %Discretized action values (just to comply with action table based on hardware capabilities)
                            end

                            [~,ii]=ismember(action_discrete(:,jji)', U, 'rows'); %finds the action ID of normalized predicted action within U


                        else %If NN failed to be trained
                            % Exploit actions from history in descending order of performance
                        end
                    end

                    %only applies for s0=1 mode (while NN have not been trained yet)
                else
                    expl=0; %flag
                    %s0 gets reset after NN is trained
                    ii=ceil(rand*length(U));%First time always explore randomly
                end

                action(jji)=ii; %Chosen action id time-series; for analysis only

                %Parameters for chosen action --> %U structure [Rs_, P_, log2(M), roll_off, M, (n/k)] for NN training
                Rs=U(ii,1);
                Es_add=U(ii,2);
                k=U(ii,3);
                r_off=U(ii,4);
                M=U(ii,5);
                rate=U(ii,6);

                %Action time series
                act_modcod_map(jji)=modcod_map(ii);
                act_Rs(jji)=U(ii,1);
                act_Es_add(jji)=U(ii,2);
                act_k(jji)=U(ii,3);
                act_r_off(jji)=r_off;
                act_M(jji)=U(ii,5);
                act_rate(jji)=U(ii,6);


                frame_dur(jji)=frame_size_/Rs; %Frame duration in secs
                elapsed_time=sum(frame_dur);

                if ceil(elapsed_time/(512/5120))> length(TOTAL2) %for analysis only (stops script if there is no more data on synthetic snr time-series)
                    break
                end


                %% Measured at Tx (sent to Rx as telemetry data):

                %BEFORE BEING AFFECTED BY CHANNEL DYNAMICS

                %Transmitted power
                measured_Es(jji)=Es_add;
                %Amplifier - Increase Es (energy per symbol)
                measured_SNR(jji)=TOTAL2(ceil(elapsed_time/(512/5120)));
                measured_SNR_lin=10^(measured_SNR(jji)/10);
                measured_SNR_lin_norm(jji)=measured_SNR_lin/max_SNR_lin;
                EsNo=measured_SNR(jji)+Es_add;

                %Consumed additional power [dB]
                measured_P_consu(jji)=10*log10((10^(Es_add/10))*Rs);
```

```
506                 measured_P_consu_lin=10^(measured_P_consu(jji)/10);
507
508                 %Power efficiency Rb/P_consu  [bits/sec/Watts]
509                 measured_Pwr_eff(jji)=(k*rate)/((10^(Es_add/10))*Rs);
510
511                 %Bandwidth
512                 measured_W(jji)=Rs*(1+r_off);% [Hz]
513
514                 %Throughput
515                 measured_T(jji)=Rs*k*rate;% [bits/sec]
516
517
518
519                 %% AI at Rx:
520                 %Measured at Rx (AFTER BEING AFFECTED BY CHANNEL DYNAMICS):
521
522                 %  This study uses SNR profiles. A real-world experiment is required to evaluate
     its performance while using real SNR measurements.
523
524                 modcod_n=modcod_map(ii); %retrieves modcod ID that maps into BER_curve function
525                 eval(sprintf('ber_func = modcod_%d;', modcod_n)) %retrieves the proper BER_curve
     function
526                 if ber_func(EsNo)<0
527                     measured_BER_est(jji)=1e-12; %assigns very low value (non-zero)
528                 elseif ber_func(EsNo)==0
529                     measured_BER_est(jji)=1e-12; %assigns very low value (non-zero)
530                 elseif ber_func(EsNo)>1
531                     measured_BER_est(jji)=1; %holds value at max = 1
532                 else
533                     measured_BER_est(jji)=ber_func(EsNo); %assigns BER value as predicted by
     function
534                 end
535
536                 %Spectral efficiency
537                 measured_spec_eff(jji)=k*rate/(1+r_off);
538
539                 %(The following are sent out by the Tx as telemetry data:)
540                 %     -Transmitted power computed at Tx + additional power used above/below link
     budget estimated for clear sky operations
541                 %     -Bandwidth computed at Tx and not affected by channel (Assumed no presence
     of interferes).
542                 %     -Roll-off factor
543
544                 %Multi-objective fitness function (f_observed reference parameters)
545
546                 % Throughput
547                 f_max_T=measured_T(jji)/T_max;
548
549                 %BER
550                 f_min_BER_true=(100-(-1*(log10(BER_min/measured_BER_est(jji))))*(-100/(log10(
     BER_min))))/100; %Function value range from 1 to BER_min scaled to 0 to 1.
551                 if f_min_BER_true>=1
552                     f_min_BER=1;
553                 else
554                     f_min_BER=f_min_BER_true;
555                 end
556
557                 % Additional power
558                 f_min_P=Es_min_lin/(10^(Es_add/10));
559
560                 %Bandwidth
561                 if measured_W(jji)<=BW %Bandwidth
562                     f_const_W=1; %No penalty if W is smaller than target BW (cannot cause
     interference)
563                 else
564                     f_const_W=1-((measured_W(jji)-BW)/BW); %If W is more than double the target
     BW
565                     if f_const_W<0
566                         f_const_W=0;
567                     end
568                 end
569
570                 %Spectral efficiency
571                 spec_eff_max=log2((mod_list(end)))*(cod_list(end))/(1+min(roll_off));
572                 f_spec_eff=measured_spec_eff(jji)/spec_eff_max;
573
574
575                 %Observed state: fitness function
576                 f_observed2(jji,:) = [((measured_T(jji)-T_min)./(T_max-T_min)) ((-10.*log10(
     measured_BER_est(jji))-ber_dB_min)./(ber_dB_max-ber_dB_min)) ((measured_W(jji)-BW_min)./(
     BW_max-BW_min)) ((measured_spec_eff(jji)-spect_eff_min)./(spect_eff_max-spect_eff_min)) ((
     log10(measured_Pwr_eff(jji))-log10(pwr_eff_min))./(log10(pwr_eff_max)-log10(pwr_eff_min)))
     1-((measured_P_consu_lin-P_consu_min_lin)./(P_consu_max_lin-P_consu_min_lin)) (
     measured_SNR_lin./max_SNR_lin)]; %range [0,1]
577                 f_observed(jji) = f_observed2(jji,1:end-1)*w'; %range [0,1] (SNR is not part of
     optimization goal)
578
579                 %Adapt/updates NN2 input whenever exploration finds a better performance
580                 if  s0==1 && hist_count==0
581                     t_e_change(cnt_debug) = jji;
582                     cnt_debug = cnt_debug + 1;
583                     e_p=f_observed(jji);
```

```
584                 elseif s0==1 && hist_count>0
585                     if f_observed(jji)>e_p
586                         t_e_change(cnt_debug) = jji;
587                         cnt_debug = cnt_debug + 1;
588                         e_p=f_observed(jji);
589                     end
590                 end
591 
592                 if f_observed(jji)>max_f_observed
593                     max_f_observed=f_observed(jji); %global best known performance so far
594                     if expl==0
595                         input_norm2=f_observed2(jji,1:end-1);
596                     end
597                 else
598                     if expl==1 % [Exploit]
599                         if f_observed(jji)<e_p
600                             if e_p-f_observed(jji)>0.1 && (sum(input_norm2==last_e_a)==length(input_norm2)) %RESET "More efficient Recover Mode". Threshold value is a designer parameter (0.5 for specific missions, 0.1 for general)
601                                 s0=1;%; %enters exploration mode
602                                 %reset NN history
603                                 hist_count=0;
604                                 temp_hist_=zeros(history_size,10);
605                                 jji_reset(jji)=jji;
606                                 max_f_observed=0;
607                             elseif f_observed(jji)<e_p*0.9 %Quick "Recover Mode" using performances from the buffer. Triggers when 90% below previous exploration level
608                                 hist2=sortrows(temp_hist_,2);
609                                 ind3=ind3+1;
610                                 if ind3==history_size
611                                     ind3=1;
612                                 end
613                                 nn2=hist2(end-ind3+1,4:9);
614                                 input_norm2=[(nn2(:,1)-T_min)./(T_max-T_min) (-10.*log10(nn2(:,2))-ber_dB_min)./(ber_dB_max-ber_dB_min) (nn2(:,3)-BW_min)./(BW_max-BW_min) (nn2(:,4)-spect_eff_min)./(spect_eff_max-spect_eff_min) (log10(nn2(:,5))-log10(pwr_eff_min))./(log10(pwr_eff_max)-log10(pwr_eff_min)) 1-((nn2(:,6)-P_consu_min_lin)./(P_consu_max_lin-P_consu_min_lin))];
615                             elseif f_observed(jji)>e_p*0.9 && ind3>0 %Accepts new exploitation performance 90% above last exploitation threshold
616                                 e_p=f_observed(jji);
617                                 last_e_a=input_norm2;
618                             else
619                                 input_norm2=last_e_a; %if exploiting and current exploitation performance is worse than previous exploitation performance; roll-back NN2 input
620                             end
621                         else
622                             e_p=f_observed(jji); %tracks last exploitation performance
623                             last_e_a=input_norm2; %tracks last exploitation NN2 input
624                         end
625                     end
626                 end
627 
628                 %Logging function measurebles; for analysis only
629                 log_f_max_T(jji)=f_max_T;
630                 log_f_min_BER(jji)=f_min_BER;
631                 log_f_min_P(jji)=f_min_P;
632                 log_f_const_W(jji)=f_const_W;
633 
634 
635                 %-->> History sliding window (shared among Explore and Exploit NN's) <<--
636                 if isempty(find(temp_hist_(:,1)==ii)) %chosen action not present in sliding window
637                     hist_count=hist_count+1; %populate
638                     if hist_count<=history_size %if sliding window not full yet
639                         ind_update=(temp_hist_(:,3)>=1);%index for update
640                         temp_hist_(hist_count,:)= [ii f_observed(jji) 1 measured_T(jji) measured_BER_est(jji) measured_W(jji) measured_spec_eff(jji) measured_Pwr_eff(jji) measured_P_consu_lin measured_SNR_lin]; % [actions f_observed time_stamp]
641                         temp_hist_(ind_update,3)=temp_hist_(ind_update,3)+1; %update all action IDs
642                     else %if sliding window already full >>> replace
643                         [~,jj]=max(temp_hist_(:,3)); %find action with oldest/highest time_stamp
644                         temp_hist_(jj,:)= [ii f_observed(jji) 0 measured_T(jji) measured_BER_est(jji) measured_W(jji) measured_spec_eff(jji) measured_Pwr_eff(jji) measured_P_consu_lin measured_SNR_lin]; %replace it
645                         temp_hist_(:,3)=temp_hist_(:,3)+1; %update all action IDs
646                     end
647                 else %chosen action is already on sliding window, update it with most recent performance
648                     ind_mean=find(temp_hist_(:,1)==ii);
649                     ind_update=(temp_hist_(:,3)>0);%index for update
650                     temp_hist_(ind_mean,:)= [ii f_observed(jji) 1 measured_T(jji) measured_BER_est(jji) measured_W(jji) measured_spec_eff(jji) measured_Pwr_eff(jji) measured_P_consu_lin measured_SNR_lin];
651                     temp_hist_(ind_update,3)=temp_hist_(ind_update,3)+1; %update all action IDs
652                 end
653 
654                 % ----> Train NN <----
655                 if hist_count==history_size % enables training after sliding window is full
656                     train_ind=[train_ind jji]; % indices when NN training took place
657 
```

```
658                     %-------------Examples NN_explore--------------
659                     % Populate with acitons (only after training window was built)
660                     hist_(:,1:size(U,2))=U((temp_hist_(:,1)),:);
661                     % Populate f_observed for non-repeating actions
662                     hist_(:,end)=temp_hist_(:,2);
663                     %Applying normalization function ps to new training input
664                     pnewn = mapminmax('apply',hist_(:,1:size(U,2))',ps);
665
666                     %NN1 training inputs
667                     examples_input= [pnewn'];
668                     examples_input=[examples_input (((temp_hist_(:,end)/max_SNR_lin)-0.5)./0.5)
    ]; %scaled to range [-1,1]
669
670                     %NN1 training outputs
671                     examples_target= [hist_(:,end)]; %range [0, 1]
672
673                     %Spliting inputs/outputs dataset (not needed for online operations)
674                     %Splits training dataset into 90% training and 10% parallel testing
675                     Q1 = floor(history_size*nn_parallel_train);
676                     Q2 = history_size-Q1; %not needed for online operations
677                     ind = randperm(history_size);
678                     ind1 = ind(1:Q1);
679                     ind2 = ind(Q1+(1:Q2)); %not needed for online operations
680                     %Training
681                     x1 = examples_input(ind1,:)';
682                     y1 = examples_target(ind1,:)';
683                     %Parallel testing (%Guarantees all nets use same test dataset)
684                     x2 = examples_input(ind2,:)'; %not needed for online operations
685                     y2 = examples_target(ind2,:)';%not needed for online operations
686
687                     %Training NN1
688                     if updateBatch
689                         tic
690                         parfor n_i=1:numNN
691                             NN{n_i}=train(net,examples_input',examples_target');
692                         end
693                         elapsedTime =toc;
694                         fprintf('Time to train explore network is %f sec. \n',elapsedTime);
695                     end
696
697                     if updateRecurse
698                         tic
699                        parfor n_i=1:numNN
700                             ind = randperm(history_size);
701                             ind1 = ind(1:Q1);
702                             ind2 = ind(Q1+1:end);
703                             x1 = examples_input(ind1,:)';
704                             y1 = examples_target(ind1,:)';
705                             x2 = examples_input(ind2,:)';
706                             y2 = examples_target(ind2,:)';
707                             [NN{n_i},NN_recurseMatrix{n_i},err(n_i)] = trainrlm(NN{n_i},
    NN_recurseMatrix{n_i},x1,y1,x2,y2);
708
709                         end
710                         elapsedTime = toc;
711                     end
712                     %flags0e
713                     NN_train=1;
714                     NN_train_2=1;
715                     first_explore=1;
716
717
718                     %-------------Examples NN_exploit--------------
719                     %NN1 training inputs
720                     examples_in_exploit = [(temp_hist_(:,4)-T_min)./(T_max-T_min) (-10.*log10(
    temp_hist_(:,5))-ber_dB_min)./(ber_dB_max-ber_dB_min) (temp_hist_(:,6)-BW_min)./(BW_max-
    BW_min) (temp_hist_(:,7)-spect_eff_min)./(spect_eff_max-spect_eff_min) (log10(temp_hist_
    (:,8))-log10(pwr_eff_min))./(log10(pwr_eff_max)-log10(pwr_eff_min)) 1-((temp_hist_(:,9)-
    P_consu_min_lin)./(P_consu_max_lin-P_consu_min_lin)) temp_hist_(:,10)./max_SNR_lin]; %range
    [0,1]
721                     examples_in_exploit=(examples_in_exploit-0.5)./0.5; %scaled to range [-1, 1]
722
723                     %NN1 training outputs
724                     examples_out_exploit = [U_out(temp_hist_(:,1),:)]; %range [0, 1]
725
726                     %Spliting inputs/outputs dataset (not needed for online operations)
727                     %Splits training dataset into 90% training and 10% parallel testing
728                     Q1 = floor(history_size*nn_parallel_train);
729                     Q2 = history_size-Q1; %not needed for online operations
730                     ind = randperm(history_size);
731                     ind1 = ind(1:Q1);
732                     ind2 = ind(Q1+(1:Q2)); %not needed for online operations
733                     %Training
734                     x1_exploit = examples_in_exploit(ind1,:)'; %not needed for online operations
735                     y1_exploit = examples_out_exploit(ind1,:)'; %not needed for online
    operations
736
737                     fprintf('training exploit network \n');
738                     tic
739                     parfor n_i = 1:numNN_exploit%parfor n_i=1:numNN_exploit
740                         for n_j=1:6
741                             ind = randperm(history_size);
```

```
742                                 ind1 = ind(1:Q1);
743                                 ind2 = ind(Q1+1:end);
744                                 x1_exploit = examples_in_exploit(ind1,:)'; %not needed for online
        operations
745                                 y1_exploit = examples_out_exploit(ind1,:)'; %not needed for online
        operations
746                                 x2_exploit = examples_in_exploit(ind2,:)';
747                                 y2_exploit = examples_out_exploit(ind2,:)';
748                                 [NN_exploit{n_j,n_i},NN_recurseMatrix_exploit{n_j,n_i}, err_exploit(
        n_j,n_i)]=trainrlm(NN_exploit{n_j,n_i},NN_recurseMatrix_exploit{n_j,n_i},x1_exploit,
        y1_exploit(n_j,:), x2_exploit,y2_exploit(n_j,:));
749                             end
750                         end
751                         elapsedTime =toc;
752                         NN_train_exploit=1;
753
754                         %-------------Reset sliding window-------------
755                         %Delete x percentage of oldest actions
756                         temp_hist_=sortrows(temp_hist_,3);
757                         hist_count=(history_size-(nn_delete*history_size)); %resets hist_count
758                         ind3=0;
759
760                         if s0==1
761                             s0=0;%reset s0; allows NN's to be used
762                         end
763                     end
764
765                 end %End of Main iteration (while)
766             end %End of multiple iterations for different channels
767             f_observed_save{mission_iter} = f_observed;
768             f_observed2_save{mission_iter} = f_observed2;
769             mission_iter = mission_iter + 1;
770         end %End of mission loop
771 end
```

C.6 CE-NSE MATLAB Simulation

```
rng('shuffle')
% clear all
close all; clc

for iterations=1

    episode_dur=21600; %only for analysis
    mission_iter = 1;
    for mission_num=[6,5,3] %[1:5] Loop over different mission profiles/fitness functions
        clearvars -except mission_num episode_dur iterations numIterations f_observed_save
     f_observed2_save  mission_iter
        f_observed=[];
        f_observed2=[];

        load('ber_functions_3') %loads DVB-S2 (Long frames) BER curve functions for online BER
     estimation using SNR measurements

        %% Build NN structures

        %-----------------------------------------
        %NN Explore (NN1)
        %Number of parallel NN
        numNN=1;
        NN = cell(1,numNN);
        nWeights = 449;
        updateBatch = 0;
        updateRecurse = 1;

        netTrain_explore=network(1,3,[0;0;0], [1 ; 0 ; 0], [ 0 0 0; 1 0 0; 0 1 0], [0 0 1]);

        %NN input size
         netTrain_explore.inputs{1}.size=1;
        %NN input range values
        netTrain_explore.inputs{1}.range = [-1 1; -1 1; -1 1; -1 1; -1 1; -1 1; -1 1];

        %NN train function
         netTrain_explore.trainFcn = 'trainlm';
        %NN dataset division function (training, validation, test)
         netTrain_explore.divideFcn='dividerand'; % 70%,15%,15% default

        % NN output functions (help nntransfer)

        % NN1 Layers
        netTrain_explore.layers{1}.size = 7;
        netTrain_explore.layers{1}.transferFcn = 'logsig';
        netTrain_explore.layers{1}.initFcn = 'initnw';%%
        netTrain_explore.layers{2}.size = 50;
        netTrain_explore.layers{2}.transferFcn = 'logsig';
        netTrain_explore.layers{2}.initFcn = 'initnw';%%
        netTrain_explore.layers{3}.size = 1;
        netTrain_explore.layers{3}.transferFcn = 'purelin';
        netTrain_explore.layers{3}.initFcn = 'initnw';%%

        netTrain_explore.trainParam.max_fail=20;
        netTrain_explore.trainParam.min_grad=1e-12;

        netTrain_explore = init(netTrain_explore);

        for i = 1:numNN

            model.type = 'MLP';          % base classifier
            NN{i}.threshold = 10e-5;          % how small is too small for error
            NN{i}.a = .5;                    % slope parameter to a sigmoid
            NN{i}.b = 10;                    % cutoff parameter to a sigmoid
            NN{i}.base_classifier = netTrain_explore;  % set the base classifier in the net
     struct
            NN{i}.classifiers = {};   % classifiers
            NN{i}.w = [1];             % weights
            NN{i}.initialized = false;% set to false
            NN{i}.t = 1;               % track the time of learning
            NN{i}.classifierweights = {};               % array of classifier weights

        end

        %Flags [do not change]
        NN_train=0; %checks if NN was trained and controls when to train it again
        NN_train_2=0;
        NN_train_exploit=0;
        %-----------------------------------------
        %NN Exploit (NN2)

        %Number of parallel NN
        numNN_exploit=1;
        NN_exploit = cell(6,numNN_exploit);
        NN_recurseMatrix_exploit = cell(6,numNN_exploit);
        nWeights_exploit = 160;

        for j = 1:6
```

```
86              netTrain_exploit=network(1,2,[0;0], [1 ; 0], [ 0 0 ; 1 0], [0 1]);
87              %NN input size
88              netTrain_exploit.inputs{1}.size=1;
89              %NN input range values
90              netTrain_exploit.inputs{1}.range = [-1 1; -1 1; -1 1; -1 1; -1 1; -1 1; -1 1];
91
92              %NN train function
93              netTrain_exploit.trainFcn = 'trainlm';
94              %NN dataset division function (training, validation, test)
95              netTrain_exploit.divideFcn='dividerand'; % 70%,15%,15% default
96
97              % NN output functions (help nntransfer)
98              netTrain_exploit.layers{1}.initFcn = 'initnw';
99              netTrain_exploit.layers{2}.initFcn = 'initnw';
100             %NN2 Layers
101             netTrain_exploit.layers{1}.size = 20;
102             netTrain_exploit.layers{1}.transferFcn = 'logsig';
103             netTrain_exploit.layers{2}.size = 1;
104             netTrain_exploit.layers{2}.transferFcn = 'purelin';
105 %             netTrain_exploit.layers{2}.transferFcn = 'logsig';
106             %Early stop conditions
107             netTrain_exploit.trainParam.max_fail=20;
108             netTrain_exploit.trainParam.min_grad=1e-12;
109             netTrain_exploit = init(netTrain_exploit);
110
111         end
112
113         for i = 1:numNN_exploit
114             for j = 1:6
115                 NN_exploit{j,i}.a = .5;                    % slope parameter to a sigmoid
116                 NN_exploit{j,i}.b = 10;                    % cutoff parameter to a sigmoid
117                 NN_exploit{j,i}.threshold = 10e-5;          % how small is too small for error
118                 NN_exploit{j,i}.base_classifier = netTrain_exploit;  % set the base classifier
    in the net struct
119                 NN_exploit{j,i}.classifiers = {};   % classifiers
120                 NN_exploit{j,i}.w = [];             % weights
121                 NN_exploit{j,i}.initialized = false;% set to false
122                 NN_exploit{j,i}.t = 1;              % track the time of learning
123                 NN_exploit{j,i}.classifierweights = {};              % array of classifier
    weights
124
125             end
126         end
127
128         %Flag
129
130         NN_train_exploit=0; %checks if NN was trained and controls when to train it again
131
132         max_f_observed=0;
133         cnt_debug = 1;
134         %% RL iterations
135
136         for iii=1:1
137             % Load Channel --> 0=GEO; 1=LEO
138             cn=1;
139
140             if cn==1
141                 % Fixed - LEO Channel [CLEAR SKY or RAIN]
142                 load ('L_fs.mat') %(LEO time series) Clear sky SNR profile at fixed ground
    receiver
143                 TOTAL=(L_fs-max(L_fs))*-1;
144             else
145                 % Fixed - GEO Channel [CLEAR SKY or RAIN]
146                 TOTAL=6*ones(1,episode_dur); %GEO clear sky SNR profile >>> 1000 seconds of
    constant 9 dB SNR profile
147             end
148             %Upsample attenuation time series to 10Hz
149             for i=1:1:length(TOTAL)
150                 TOTAL2(10*i-9:10*i)=TOTAL(i);
151             end
152
153             %% Initializing variables
154
155             %Adaptation parameters
156
157             % IN CASE **ANY** PARAMETER CHANGE ITS RANGE [MIN, MAX] the
158             % NORMALIZATION 'ps' function MUST BE LEARNED again !!!
159
160             mod_list = [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 8, 8, 8, 8, 8, 8, 16, 16, 16, 16, 16,
    16, 32, 32, 32, 32, 32];
161             cod_list = [1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 8/9, 9/10, 3/5, 2/3, 3/4,
    5/6, 8/9, 9/10, 2/3, 3/4, 4/5, 5/6, 8/9, 9/10, 3/4, 4/5, 5/6, 8/9, 9/10];
162
163             M_=[4 8 16 32];
164             k_=log2(M_);
165
166             BW_max=5e6; %MHz --> Max single-frequency SCaN testbed transponder bandwidth
167             BW_min=0.5e6; %MHz
168
169             roll_off=[0.2 0.3 0.35]; %Squared-root raised-cosine filter roll-off factor
170
171             Rs_max=BW_max/(1+max(roll_off)); %Max Symbol rate such that BW_max value is not
```

```
compromised
172         Rs_min=BW_min/(1+min(roll_off));
173
174         T_max= Rs_max * log2(max(mod_list)) * max(cod_list); %Throughput in bits/sec
175         T_min= Rs_min * log2(min(mod_list)) * min(cod_list);
176
177         frame_size_=64800;% in bits --->Long-frame DVB-S2
178
179         % Ranges of adaptable parameters for value scalling of NN inputs
180         modcod_=1:length(mod_list); %Mod + Cod
181         Rs_=Rs_min:0.1*1e6:Rs_max;     %Symbol rate range
182         Es_=0:10; %Additional Es/No [dB] to boost signal before channel;
183
184         %Values used for normalization of monitored parameters
185         Es_min_lin=10^(min(Es_)/10);
186         Es_max_lin=10^(max(Es_)/10);
187
188         %Consumed Power
189         P_consu_min_lin=(Es_min_lin*Rs_min); %linear
190         P_consu_max_lin=(Es_max_lin*Rs_max);
191
192         %Spectral efficiency
193         spect_eff_max=max(log2(mod_list))*max(cod_list)/(1+min(roll_off));
194         spect_eff_min=min(log2(mod_list))*min(cod_list)/(1+max(roll_off));
195
196         %Consumed power efficiency
197         pwr_eff_max=(max(log2(mod_list))*max(cod_list))/(Es_min_lin*Rs_min);
198         pwr_eff_min=(min(log2(mod_list))*min(cod_list))/(Es_max_lin*Rs_max);
199
200         %SNR
201         max_SNR=12.9263; % [dB] Maximum link margin achieved during clear sky conditions for
     current predicted orbit (obtained from link budget)
202         max_SNR_lin=10^(max_SNR/10);
203
204         %Designer paremeters: comms target performance BER value
205         BER_max=1e-12;% max BER
206         ber_dB_max=-10.*log10(BER_max);
207         ber_dB_min=-10.*log10(1);
208
209         BER_min=1e-6;% min BER
210
211         BW=1e6; % Constant bandwidth [Hz]
212
213         %   -->AI parameters<--
214
215         %RL parameters
216         tr=0;% RL state reward threshold
217         % [no (gamma) discount factor ->REMOVED TEMPORAL DIFFERENCE]
218         episilon(1)=1;
219
220         %U matrix - all parameter combinations between Rs_,Es_,modcod_,roll_off
221         U=(combvec(Rs_,Es_,modcod_,roll_off))';
222
223         %Add additional features for NN training
224         U=[U zeros(size(U,1),1) zeros(size(U,1),1)];
225         U(:,5)=mod_list(U(:,3));
226         U(:,6)=cod_list(U(:,3));
227         modcod_map=U(:,3);
228         U(:,3)=log2(U(:,5)); %replaces modcod mapping by mod_order
229
230         %Final U structure for NN training [Rs_, Es_, log2(M), roll_off, M, (n/k)]
231
232         %Get Actions normalization function (rows are features). Used to build normalized
    training set
233         [~,ps] = mapminmax(U');
234
235         %Scaled action matrix U_out, range [0, 1]
236         U_out=U;
237         for s=1:size(U,2)
238             U_out(:,s)=(U_out(:,s)-min(U_out(:,s)))/(max(U_out(:,s))-min(U_out(:,s))); %
    range [0,1]
239         end
240
241         %List to classify predicted modcod (used at NN2 output)
242         a_=unique(U_out(:,3));
243         b_=unique(U_out(:,5));
244         x=0;
245         for i=1:4
246             for j=1:4
247                 x=x+1;
248                 sum_(x)=a_(i)+b_(j); %[0 0.14 0.33 0.44 0.47 0.66 0.77 0.8 1 1 1.1 1.14 1.33
     1.44 1.66 2] Classification targets are: [0 .4762 1.0952 2]
249             end
250         end
251
252         s0=1;% Initialization control to start as random exploration until NN gets trained
253
254         %Communication mission phases/scenarios
255         %     [Thrp, BER, BW, Spec_eff, Pwr_eff, Pwr ];
256         mission =...
257             [0.2   0.4   0.1    0.1    0.1    0.1;      %Launch/re-entry (1)
258              0.5   0.3   0.05   0.05   0.05   0.05;     %Multimedia      (2)
```

```
259                 0.05  0.05  0.05   0.05   0.3    0.5;      %Power saving     (3)
260                 1/6   1/6   1/6    1/6    1/6    1/6;      %Normal           (4)
261                 0.05  0.05  0.4    0.4    0.05   0.05;     %Cooperation      (5)
262                 0.1   0.8   0.025  0.025  0.025  0.025];   %Emergency        (6)
263
264         w=mission(mission_num,:); %Select communication mission.
265
266         n=2; %Initial exploration probability divider
267
268         %NO Q-table anymore! :)
269
270         %%  MAIN SIMULATION ITERATION
271
272
273         episilon_reset_lim=4e-3; %reset episilon when episilon < episilon_reset_lim
274         %Log observables (for performance analysis only)
275         log_f_max_T=zeros(1,episode_dur);
276         log_f_min_BER=zeros(1,episode_dur);
277         log_f_min_P=zeros(1,episode_dur);
278         log_f_const_W=zeros(1,episode_dur);
279
280         %Normalizing all NN exploration inputs (actions)
281         input_explore_2 = mapminmax('apply',U',ps); %Applying normalization function ps to
    new training input
282
283         %Percentage to exclude from training buffer for next retraining
284         nn_delete=1;%1/4; %after training, delete 50% of oldest actions and retrain only
    after these percentage of training data is replaced by new training data
285
286         history_size=200; % <<<<< NN training window with UNIQUE actions (and its respective
     performance) >>> designer parameter ANALYZE IMPACT ON PERFORMANCE!!!!!
287
288         % Percentage of training data used for training [for online learning 100% history
    data can be used] (remaining training data is used for parellel testing)
289         nn_parallel_train=1; %(90% for training and 10% for parellel testing)
290
291         hist_count=0;
292         ind_mean=0;
293
294         temp_hist_=zeros(history_size,10); %Sliding window of last action-parformance data [
    acdionID f_observed time_stamp]
295         hist_=zeros(history_size,size(U,2)+1); %[action_parameters f_observed]
296
297         action_pred=zeros(1,episode_dur);
298         action_discrete=zeros(1,episode_dur);
299
300         %NN1 threshold prediction values for virtual exploration
301         perc_max_explore=.90; % threshold percentage of current achievable maximum predicted
     performance
302         rejection_rate=.95; %rejection percentage (rate) of exploring actions which
    performance predictions fall below the performance_threshold value
303
304         train_ind=[];% indices when NN training took place
305
306         track_exploit_err=0;
307
308         explore_control=0; %used in while loop during explore_mode=1 ONLY (starts as 0
    always to be run once)
309
310         first_explore=0;
311
312         e_p=[]; %exploit performance (holds last exploitation performance value; if new/
    current exploitation action resuls in poorer performance than previous, roll-back last
    exploitation action
313         last_e_a=[NaN NaN NaN NaN NaN NaN];
314
315         elapsed_time=0; %time counter [seconds]
316         jji=0; %main discrete time index/packet counter
317         frame_dur=[];
318
319         ind3=0;
320
321         %% Main processing loop basedd on time duration (tracked by elapsed time, computing
    packet duration)
322         while elapsed_time<512%length(time_series(3).esno_viasat)%elapsed_time<512%512
323             jji=jji+1; %increments packet counter
324
325             % AI functions
326             % [For implementation include parameter change monitoring feature]
327             % If [Rs_, Es_, log2(M), roll_off, M, (n/k)] (either max, min, or any value
    within its ranges) changes, do the following:
328             %   -Updates matrix U in case adaptable parameters changed
329             %   -Update reference parameters for scaling
330
331             %Compute epsilon(jji) %e-greedy function of time
332             %Decrease exploration rate/increase exploitation
333             if jji>=(history_size + 1) && episilon(jji-1)> episilon_reset_lim
334                 episilon(jji)=1/n; % n is # os times system exploited config. After episilon
    (jji-1)> episilon_reset_lim, reset exploration probability.
335                 n=n+1;
336             else
337                 episilon(jji)=1;
```

```
338                     n=1;%reset epsilon
339                 end
340
341
342                 %-->> (Exploit or Explore) epsilon=Exploration probability <<--%
343                 if s0==0
344                     e_prob=rand; %Randomly (uniformly) pick Explore or Exploit
345
346                     % [Explore]
347                     if e_prob<=episilon(jji)
348                         expl=0; %flag
349                         fprintf('Explore \n');
350                         %-----NN1 Prediction--->Exploration
351                         %Predictions are done every iteration ONLY after NN is trained and ONLY
       if exploring
352                         if NN_train==1
353                             %Inputs
354                             input_explore=[input_explore_2; (ones(1,length(U)))*(((
       measured_SNR_lin/max_SNR_lin)-0.5)./0.5)]; %scaling to range [-1, 1]
355
356                             %%% USAGE OF NNs
357                             %Predictions
358
359                             parfor i_NN=1:numNN
360                                 numOutputs = size(input_explore,2);
361                                 numExperts = size(NN{i_NN}.classifiers,2);
362                                 y_tmp = regress_ensemble(NN{i_NN},input_explore,numExperts);
363                                 y_pred(i_NN,:) = y_tmp;
364                             end
365                             if size(y_pred,1) > 1
366                                f_predic=mean(y_pred); %Ensamble average prediction
367                             else
368                                 f_predic = y_pred;
369                             end
370
371                             %Logic to decide on actions predicted by NN
372                             perf_th=max(f_predic)*perc_max_explore; %current performance
       threshold
373
374                             %get indexes of f_predic >= perf_th
375                             bigger_f_predic=find(f_predic>=perf_th);
376                             %get indexes of f_predic < perf_th
377                             smaller_f_predic=find(f_predic<perf_th);
378
379                             NN_train_2=0;
380
381                             if first_explore==1%right after training the action with max
       predicted performance is explored first (could guide Exploitation NN input)
382                                 ii=find(f_predic==max(f_predic));
383                                 ii=datasample(ii,1);
384                                 first_explore=0;
385                             else
386                                 if rand>=rejection_rate
387                                     % pick random action with performance < perf_th
388                                     if not(isempty(smaller_f_predic))
389                                         ii=datasample(smaller_f_predic,1);
390                                     else
391                                         ii=datasample(bigger_f_predic,1);
392                                     end
393                                 else
394                                     %pick random action with performance > perf_th
395                                     if not(isempty(bigger_f_predic))
396                                         ii=datasample(bigger_f_predic,1);
397                                     else
398                                         ii=datasample(smaller_f_predic,1);
399                                     end
400                                 end
401                             end
402                         end
403 %                         ii
404                         track_exploit_err=0; %reset when explore
405
406
407                     % [Exploit]
408                     else
409                         expl=1; %flag
410
411                         %-----NN2 Prediction--->Exploitation
412                         %Predictions are done every iteration ONLY after NN is trained and ONLY
       if exploring
413                         if NN_train_exploit==1 %Use NN predictions if NN has been already
       trained
414
415                             %Inputs
416                             input_norm=[input_norm2 (measured_SNR_lin/max_SNR_lin)]';
417                             input_norm=(input_norm-0.5)./0.5; %scaling to range [-1, 1]
418
419                             %%% NN_exploit usage
420                             %Predictions
421
422                             for n_i=1:numNN_exploit
423                                 parfor n_j=1:6 % parfor
```

```
424                                     numExperts_exploit = size(NN_exploit{n_j,n_i}.classifiers,2)
    ;
425                                     norm_action_pred(n_i,n_j) = regress_ensemble(NN_exploit{n_j,
    n_i},input_norm,numExperts_exploit);
426                                 end
427                             end
428                             if size(norm_action_pred,1) > 1
429                                norm_action=mean(norm_action_pred);%Ensamble average prediction
430                             else
431                                 norm_action = norm_action_pred;
432                             end
433
434                             %Classify modcod
435                             [~,modcod_class]=min(abs(sum_(1,1:5:16)-(norm_action(3)+norm_action
    (5))));
436
437                             %Denormalize predicted Action
438                             for s=1:size(U,2)
439                                 action_pred(s,jji)=(norm_action(s)*(max(U(:,s))-min(U(:,s))))+
    min(U(:,s)); %only for simulation analysis
440
441                                 %Classify denormalized values into executable action parameters
    [Action structure [Rs_, Es_, log2(M), roll_off, M, (n/k)] for NN training]
442                                 %Switcting between vectors (edges)
443                                 if s==1
444                                     edges=Rs_;
445                                 elseif s==2
446                                     edges=Es_;
447                                 elseif s==3
448                                     edges=2:5; %log2(M)
449                                 elseif s==4
450                                     edges=roll_off;
451                                 elseif s==5
452                                     edges=M_;
453                                 elseif s==6 %classify encoding rate based on the modulation
    order classified previously
454                                     if (action_discrete(5,jji))==4
455                                         edges=[1/4 1/3 2/5 1/2 3/5 2/3 3/4 4/5 5/6 8/9 9/10];
456                                     elseif (action_discrete(5,jji))==8
457                                         edges=[3/5 2/3 3/4 5/6 8/9 9/10];
458                                     elseif (action_discrete(5,jji))==16
459                                         edges=[2/3 3/4 4/5 5/6 8/9 9/10];
460                                     elseif (action_discrete(5,jji))==32
461                                         edges=[3/4 4/5 5/6 8/9 9/10];
462                                     end
463                                 end
464
465                                 if s==3 || s==5
466                                     min_ind=modcod_class;
467                                 else
468                                     [~,min_ind]=min(abs(action_pred(s,jji)-edges)); %
    clasification using minimum distance
469                                 end
470                                 action_discrete(s,jji)=edges(min_ind); %Discretized action
    values (just to comply with action table based on hardware capabilities)
471                             end
472
473                             [~,ii]=ismember(action_discrete(:,jji)', U, 'rows'); %finds the
    action ID of normalized predicted action within U
474
475
476                         else %If NN failed to be trained
477                             % Exploit actions from history in descending order of performance
478                         end
479                     end
480
481                     %only applies for s0=1 mode (while NN have not been trained yet)
482                 else
483                     expl=0; %flag
484                     %s0 gets reset after NN is trained
485                     ii=ceil(rand*length(U));%First time always explore randomly
486                 end
487
488                 action(jji)=ii; %Chosen action id time-series; for analysis only
489
490                 %Parameters for chosen action --> %U structure [Rs_, P_, log2(M), roll_off, M, (
    n/k)] for NN training
491                 Rs=U(ii,1);
492                 Es_add=U(ii,2);
493                 k=U(ii,3);
494                 r_off=U(ii,4);
495                 M=U(ii,5);
496                 rate=U(ii,6);
497
498                 %Action time series
499                 act_modcod_map(jji)=modcod_map(ii);
500                 act_Rs(jji)=U(ii,1);
501                 act_Es_add(jji)=U(ii,2);
502                 act_k(jji)=U(ii,3);
503                 act_r_off(jji)=r_off;
504                 act_M(jji)=U(ii,5);
505                 act_rate(jji)=U(ii,6);
```

```
506
507
508                 frame_dur(jji)=frame_size_/Rs; %Frame duration in secs
509                 elapsed_time=sum(frame_dur);
510
511                 if ceil(elapsed_time/(512/5120))> length(TOTAL2) %for analysis only (stops
    script if there is no more data on synthetic snr time-series)
512                     break
513                 end
514
515
516                 %% Measured at Tx (sent to Rx as telemetry data):
517
518                 %BEFORE BEING AFFECTED BY CHANNEL DYNAMICS
519
520                 %Transmitted power
521                 measured_Es(jji)=Es_add;
522                 %Amplifier - Increase Es (energy per symbol)
523                 measured_SNR(jji)=TOTAL2(ceil(elapsed_time/(512/5120)));
524                 measured_SNR_lin=10^(measured_SNR(jji)/10);
525                 measured_SNR_lin_norm(jji)=measured_SNR_lin/max_SNR_lin;
526                 EsNo=measured_SNR(jji)+Es_add;
527
528                 %Consumed additional power [dB]
529                 measured_P_consu(jji)=10*log10((10^(Es_add/10))*Rs);
530                 measured_P_consu_lin=10^(measured_P_consu(jji)/10);
531
532                 %Power efficiency Rb/P_consu  [bits/sec/Watts]
533                 measured_Pwr_eff(jji)=(k*rate)/((10^(Es_add/10))*Rs);
534
535                 %Bandwidth
536                 measured_W(jji)=Rs*(1+r_off);% [Hz]
537
538                 %Throughput
539                 measured_T(jji)=Rs*k*rate;% [bits/sec]
540
541
542
543                 %% AI at Rx:
544                 %Measured at Rx (AFTER BEING AFFECTED BY CHANNEL DYNAMICS):
545
546                 %  This study uses SNR profiles. A real-world experiment is required to evaluate
     its performance while using real SNR measurements.
547
548                 modcod_n=modcod_map(ii); %retrieves modcod ID that maps into BER_curve function
549                 eval(sprintf('ber_func = modcod_%d;', modcod_n)) %retrieves the proper BER_curve
     function
550                 if ber_func(EsNo)<0
551                     measured_BER_est(jji)=1e-12; %assigns very low value (non-zero)
552                 elseif ber_func(EsNo)==0
553                     measured_BER_est(jji)=1e-12; %assigns very low value (non-zero)
554                 elseif ber_func(EsNo)>1
555                     measured_BER_est(jji)=1; %holds value at max = 1
556                 else
557                     measured_BER_est(jji)=ber_func(EsNo); %assigns BER value as predicted by
    function
558                 end
559
560                 %Spectral efficiency
561                 measured_spec_eff(jji)=k*rate/(1+r_off);
562
563                 %(The following are sent out by the Tx as telemetry data:)
564                 %     -Transmitted power computed at Tx + additional power used above/below link
     budget estimated for clear sky operations
565                 %     -Bandwidth computed at Tx and not affected by channel (Assumed no presence
     of interferes).
566                 %     -Roll-off factor
567
568                 %Multi-objective fitness function (f_observed reference parameters)
569
570                 % Throughput
571                 f_max_T=measured_T(jji)/T_max;
572
573                 %BER
574                 f_min_BER_true=(100-(-1*(log10(BER_min/measured_BER_est(jji))))*(-100/(log10(
    BER_min))))/100; %Function value range from 1 to BER_min scaled to 0 to 1.
575                 if f_min_BER_true>=1
576                     f_min_BER=1;
577                 else
578                     f_min_BER=f_min_BER_true;
579                 end
580
581                 % Additional power
582                 f_min_P=Es_min_lin/(10^(Es_add/10));
583
584                 %Bandwidth
585                 if measured_W(jji)<=BW %Bandwidth
586                     f_const_W=1; %No penalty if W is smaller than target BW (cannot cause
    interference)
587                 else
588                     f_const_W=1-((measured_W(jji)-BW)/BW); %If W is more than double the target
    BW
```

```
589                     if f_const_W<0
590                         f_const_W=0;
591                     end
592                 end
593
594                 %Spectral efficiency
595                 spec_eff_max=log2((mod_list(end)))*(cod_list(end))/(1+min(roll_off));
596                 f_spec_eff=measured_spec_eff(jji)/spec_eff_max;
597
598
599                 %Observed state: fitness function
600                 f_observed2(jji,:) = [((measured_T(jji)-T_min)./(T_max-T_min)) ((-10.*log10(
    measured_BER_est(jji))-ber_dB_min)./(ber_dB_max-ber_dB_min)) ((measured_W(jji)-BW_min)./(
    BW_max-BW_min)) ((measured_spec_eff(jji)-spect_eff_min)./(spect_eff_max-spect_eff_min)) ((
    log10(measured_Pwr_eff(jji))-log10(pwr_eff_min))./(log10(pwr_eff_max)-log10(pwr_eff_min)))
    1-((measured_P_consu_lin-P_consu_min_lin)./(P_consu_max_lin-P_consu_min_lin)) (
    measured_SNR_lin./max_SNR_lin)]; %range [0,1]
601                 f_observed(jji) = f_observed2(jji,1:end-1)*w'; %range [0,1] (SNR is not part of
    optimization goal)
602
603                 %Adapt/updates NN2 input whenever exploration finds a better performance
604                 if  s0==1 && hist_count==0
605                     fprintf('e_p changes, f1, %f\n', f_observed(jji));
606                     t_e_change(cnt_debug) = jji;
607                     cnt_debug = cnt_debug + 1;
608                     e_p=f_observed(jji);
609                 elseif s0==1 && hist_count>0
610                     if f_observed(jji)>e_p
611                         fprintf('e_p changes,f2, %f\n', f_observed(jji));
612                         t_e_change(cnt_debug) = jji;
613                         cnt_debug = cnt_debug + 1;
614                         e_p=f_observed(jji);
615                     end
616                 end
617
618                 if f_observed(jji)>max_f_observed
619                     max_f_observed=f_observed(jji); %global best known performance so far
620                     if expl==0
621                         input_norm2=f_observed2(jji,1:end-1);
622                     end
623                 else
624                     if expl==1 % [Exploit]
625                         if f_observed(jji)<e_p
626                             if e_p-f_observed(jji)>0.1 && (sum(input_norm2==last_e_a)==length(
    input_norm2)) %RESET "More efficient Recover Mode". Threshold value is a designer parameter
    (0.5 for specific missions, 0.1 for general)
627                                 s0=1;%; %enters exploration mode
628                                 %reset NN history
629                                 hist_count=0;
630                                 temp_hist_=zeros(history_size,10);
631                                 jji_reset(jji)=jji;
632                                 max_f_observed=0;
633                             elseif f_observed(jji)<e_p*0.9 %Quick "Recover Mode" using
    performances from the buffer. Triggers when 90% below previous exploration level
634                                 hist2=sortrows(temp_hist_,2);
635                                 ind3=ind3+1;
636                                 if ind3==history_size
637                                     ind3=1;
638                                 end
639                                 nn2=hist2(end-ind3+1,4:9);
640                                 input_norm2=[(nn2(:,1)-T_min)./(T_max-T_min) (-10.*log10(nn2
    (:,2))-ber_dB_min)./(ber_dB_max-ber_dB_min) (nn2(:,3)-BW_min)./(BW_max-BW_min) (nn2(:,4)-
    spect_eff_min)./(spect_eff_max-spect_eff_min) (log10(nn2(:,5))-log10(pwr_eff_min))./(log10(
    pwr_eff_max)-log10(pwr_eff_min)) 1-((nn2(:,6)-P_consu_min_lin)./(P_consu_max_lin-
    P_consu_min_lin))];
641                             elseif f_observed(jji)>e_p*0.9 && ind3>0 %Accepts new exploitation
    performance 90% above last exploitation threshold
642                                 e_p=f_observed(jji);
643                                 last_e_a=input_norm2;
644                             else
645                                 input_norm2=last_e_a; %if exploiting and current exploitation
    performance is worse than previous exploitation performance; roll-back NN2 input
646                             end
647                         else
648                             e_p=f_observed(jji); %tracks last exploitation performance
649                             last_e_a=input_norm2; %tracks last exploitation NN2 input
650                         end
651                     end
652                 end
653
654                 %Logging function measurebles; for analysis only
655                 log_f_max_T(jji)=f_max_T;
656                 log_f_min_BER(jji)=f_min_BER;
657                 log_f_min_P(jji)=f_min_P;
658                 log_f_const_W(jji)=f_const_W;
659
660
661                 %-->> History sliding window (shared among Explore and Exploit NN's) <<--
662                 if isempty(find(temp_hist_(:,1)==ii)) %chosen action not present in sliding
    window
663                     hist_count=hist_count+1; %populate
664                     if hist_count<=history_size %if sliding window not full yet
```

```
665                         ind_update=(temp_hist_(:,3)>=1);%index for update
666                         temp_hist_(hist_count,:)= [ii f_observed(jji) 1 measured_T(jji)
    measured_BER_est(jji) measured_W(jji) measured_spec_eff(jji) measured_Pwr_eff(jji)
    measured_P_consu_lin measured_SNR_lin]; % [actions f_observed time_stamp]
667                         temp_hist_(ind_update,3)=temp_hist_(ind_update,3)+1; %update all action
    IDs
668                     else %if sliding window already full >>> replace
669                         [~,jj]=max(temp_hist_(:,3)); %find action with oldest/highest time_stamp
670                         temp_hist_(jj,:)= [ii f_observed(jji) 0 measured_T(jji) measured_BER_est
    (jji) measured_W(jji) measured_spec_eff(jji) measured_Pwr_eff(jji) measured_P_consu_lin
    measured_SNR_lin]; %replace it
671                         temp_hist_(:,3)=temp_hist_(:,3)+1; %update all action IDs
672                     end
673                 else %chosen action is already on sliding window, update it with most recent
    performance
674                     ind_mean=find(temp_hist_(:,1)==ii);
675                     ind_update=(temp_hist_(:,3)>0);%index for update
676                     temp_hist_(ind_mean,:)= [ii f_observed(jji) 1 measured_T(jji)
    measured_BER_est(jji) measured_W(jji) measured_spec_eff(jji) measured_Pwr_eff(jji)
    measured_P_consu_lin measured_SNR_lin];
677                     temp_hist_(ind_update,3)=temp_hist_(ind_update,3)+1; %update all action IDs
678                 end
679
680                 % ----> Train NN <---
681                 if hist_count==history_size % enables training after sliding window is full
682                     train_ind=[train_ind jji]; % indices when NN training took place
683
684                     %-------------Examples NN_explore--------------
685                     % Populate with acitons (only after training window was built)
686                     hist_(:,1:size(U,2))=U((temp_hist_(:,1)),:);
687                     % Populate f_observed for non-repeating actions
688                     hist_(:,end)=temp_hist_(:,2);
689                     %Applying normalization function ps to new training input
690                     pnewn = mapminmax('apply',hist_(:,1:size(U,2))',ps);
691
692                     %NN1 training inputs
693                     examples_input= [pnewn'];
694                     examples_input=[examples_input (((temp_hist_(:,end)/max_SNR_lin)-0.5)./0.5)
    ]; %scaled to range [-1,1]
695
696                     %NN1 training outputs
697                     examples_target= [hist_(:,end)]; %range [0, 1]
698
699                     %Spliting inputs/outputs dataset (not needed for online operations)
700                     %Splits training dataset into 90% training and 10% parallel testing
701                     Q1 = floor(history_size*nn_parallel_train*0.9);
702                     Q2 = history_size-Q1; %not needed for online operations
703                     ind = randperm(history_size);
704                     ind1 = ind(1:Q1);
705                     ind2 = ind(Q1+(1:Q2)); %not needed for online operations
706                     %Training
707                     x1 = examples_input(ind1,:)';
708                     y1 = examples_target(ind1,:)';
709                     %Parallel testing (%Guarantees all nets use same test dataset)
710                     x2 = examples_input(ind2,:)'; %not needed for online operations
711                     y2 = examples_target(ind2,:)';%not needed for online operations
712
713
714                     %Training NN1
715                     if updateBatch
716                         tic
717                         parfor n_i=1:numNN
718                             NN{n_i}=train(net,x1,y1);
719                         end
720                         elapsedTime =toc;
721                         fprintf('Time to train explore network is %f sec. \n',elapsedTime);
722                     end
723
724                     if updateRecurse
725                         tic
726                        parfor i = 1:numNN
727                             [NN{i},~,~,~,~,err] = ...
728                                 learn_nse_update(NN{i}, x1, y1, x2, ...
729                                                  y2);
730                         end
731                         elapsedTime = toc;
732                     end
733                     %flags
734                     NN_train=1;
735                     NN_train_2=1;
736                     first_explore=1;
737
738
739                     %-------------Examples NN_exploit--------------
740                     %NN1 training inputs
741                     examples_in_exploit = [(temp_hist_(:,4)-T_min)./(T_max-T_min) (-10.*log10(
    temp_hist_(:,5))-ber_dB_min)./(ber_dB_max-ber_dB_min) (temp_hist_(:,6)-BW_min)./(BW_max-
    BW_min) (temp_hist_(:,7)-spect_eff_min)./(spect_eff_max-spect_eff_min) (log10(temp_hist_
    (:,8))-log10(pwr_eff_min))./(log10(pwr_eff_max)-log10(pwr_eff_min)) 1-((temp_hist_(:,9)-
    P_consu_min_lin)./(P_consu_max_lin-P_consu_min_lin)) temp_hist_(:,10)./max_SNR_lin]; %range
    [0,1]
742                     examples_in_exploit=(examples_in_exploit-0.5)./0.5; %scaled to range [-1, 1]
```

```
743
744                     %NN1 training outputs
745                     examples_out_exploit = [U_out(temp_hist_(:,1),:)]; %range [0, 1]
746
747                     %Spliting inputs/outputs dataset (not needed for online operations)
748                     %Splits training dataset into 90% training and 10% parallel testing
749                     Q1 = floor(history_size*nn_parallel_train * 0.9);
750                     Q2 = history_size-Q1; %not needed for online operations
751                     ind = randperm(history_size);
752                     ind1 = ind(1:Q1);
753                     ind2 = ind(Q1+(1:Q2)); %not needed for online operations
754                     %Training
755                     x1_exploit = examples_in_exploit(ind1,:)'; %not needed for online operations
756                     y1_exploit = examples_out_exploit(ind1,:)'; %not needed for online
       operations
757                     %Training NN2
758                     tic
759                     for n_i=1:numNN_exploit % parfor
760                         for n_j=1:6
761                             ind = randperm(history_size);
762                             ind1 = ind(1:Q1);
763                             ind2 = ind(Q1:end);
764                             x1_exploit = examples_in_exploit(ind1,:)'; %not needed for online
       operations
765                             y1_exploit = examples_out_exploit(ind1,:)'; %not needed for online
       operations
766                             x2_exploit = examples_in_exploit(ind2,:)';
767                             y2_exploit = examples_out_exploit(ind2,:)';
768 %                               NN_exploit{n_j,n_i}=train(net_exploit,x1_exploit,y1_exploit(n_j,:)
       ); %training per feeature NN
769                             [NN_exploit{n_j,n_i},~,~,~,err_exploit] = ...
770                                 learn_nse_update(NN_exploit{n_j,n_i}, x1_exploit, y1_exploit(n_j
       ,:), x2_exploit, ...
771                                                                 y2_exploit(n_j,:));
772
773                         end
774                     end
775                     elapsedTime =toc;
776                     NN_train_exploit=1;
777
778                     %--------------Reset sliding window-------------
779                     %Delete x percentage of oldest actions
780                     temp_hist_=sortrows(temp_hist_,3);
781                     temp_hist_((history_size-(nn_delete*history_size))+1:end,:)=0;
782                     hist_count=(history_size-(nn_delete*history_size)); %resets hist_count
783                     ind3=0;
784
785                     if s0==1
786                         s0=0;%reset s0; allows NN's to be used
787                     end
788                 end
789
790             end %End of Main iteration (while)
791         end %End of multiple iterations for different channels
792         f_observed_save{mission_iter} = f_observed;
793         f_observed2_save{mission_iter} = f_observed2;
794         mission_iter = mission_iter + 1;
795     end %End of mission loop
796 end
```

C.7 RecurseMatrix.m

```
classdef RecurseMatrix < handle
    properties
        Gradient
        Time
        P
        rho
        S
        alpha
        sizeBatch
    end
    methods
        function obj = RecurseMatrix(grad,time,sizeBatch,alpha)
            obj.Gradient = grad;
            obj.Time = time;
            obj.P = eye(sizeBatch) ;%.* rand(sizeBatch,sizeBatch);%./sizeBatch;%randn(sizeBatch,
    sizeBatch);
            obj.S = eye(2);
            obj.rho = 100;
            obj.alpha = alpha;
            obj.sizeBatch = sizeBatch;

        end
        function obj = updateMatrix(obj,grad,time,PMat,SMat,newRho,newAlpha)
            obj.Gradient = grad;
            obj.Time = time;
            obj.P = PMat;%.* rand(sizeBatch,sizeBatch);%./sizeBatch;%randn(sizeBatch,sizeBatch);
            obj.S = SMat;
            obj.rho = newRho;
            obj.alpha = newAlpha;

        end
    end
end
```

C.8 decision_ensemble.m

```
function y = decision_ensemble(net, data, labels, n_experts)
if numel(labels) ~= 1
    y = zeros(numel(labels), n_experts);
else
    y = zeros(labels,n_experts);
end

for k = 1:n_experts
  y(:, k) = net.classifiers{k}(data);
end
if n_experts > 1
    y = mean(y);
end
```

C.9 learn_nse_update.m

```
function  [netOut,f_measure,g_mean,recall,precision,...
    err] = learn_nse_update(net, data_train, labels_train, data_test, ...
  labels_test)
numClassifiers = 20;
  if net.initialized == false, net.beta = []; end

  mt = size(data_train,2); % number of training examples
  Dt = ones(mt,1)/mt;         % initialize instance weight distribution
  Dt_sampBySamp = Dt;
  if net.initialized==1
    % STEP 1: Compute error of the existing ensemble on new data
    predictions = regress_ensemble(net, data_train, labels_train);

    Et_sampBySamp = mse_reg(predictions.',labels_train)/mt;
    Et = sum(Et_sampBySamp);
    %%% get a beta for each net.
    Bt = Et/(1-Et);           % this is suggested in Metin's IEEE Paper
    Bt_sampBySamp = Et_sampBySamp./(1-Et_sampBySamp);
    if Bt==0, Bt = 1/mt; end; % clip

    % update and normalize the instance weights
    Wt = 1/mt * Bt;
    Dt = Wt/sum(Wt);
    Wt_sampBySamp = 1/mt * Bt_sampBySamp;
    Dt_sampBySamp = Wt_sampBySamp/sum(Wt_sampBySamp);
  end

  % STEP 3: New classifier
  if size(net.classifiers,2) < numClassifiers
      net.classifiers{end + 1} = train(...
        net.base_classifier, ...
        data_train, ...
        labels_train);
  else %%% TODO: Proper pruning instead of oldest-out
      net.classifiers{mod(size(net.classifiers,2),20) + 1} = train(...
        net.base_classifier, ...
        data_train, ...
        labels_train);
  end

  % STEP 4: Evaluate all existing classifiers on new data
  t = size(net.classifiers,2);
  y = decision_ensemble(net, data_train, labels_train, t);
  for k = 1:min(net.t,numClassifiers)
    epsilon_tk = sum(Dt_sampBySamp.*mse_reg(y(:,k),labels_train.')/mt);
    if (k<net.t)&&(epsilon_tk>0.5)
      epsilon_tk = 0.5;
    elseif (k==net.t)&&(epsilon_tk>0.5)
      % try generate a new classifier
      net.classifiers{k} = train(...
        net.base_classifier, ...
        data_train, ...
        labels_train);
      epsilon_tk  = sum(Dt(y(:, k) ~= labels_train));
      epsilon_tk(epsilon_tk > 0.5) = 0.5;   % we tried; clip the loss
    end
    net.beta(net.t,k) = epsilon_tk / (1-epsilon_tk);
  end

  % compute the classifier weights
  if net.t==1
    if net.beta(net.t,net.t)<net.threshold
      net.beta(net.t,net.t) = net.threshold;
    end
    net.w(net.t,net.t) = log(1/net.beta(net.t,net.t));
  else
    for k = 1:min(net.t,numClassifiers)
      b = t - k - net.b;
      if net.t <= numClassifiers
        omega = 1:(net.t - k + 1);
      else
          omega = 1:numClassifiers;
      end
      omega = 1./(1+exp(-net.a*(omega-b)));
      omega = (omega/sum(omega))';
      if net.t <= numClassifiers
        beta_hat = sum(omega.*(net.beta(k:net.t,k)));
      else
          net.beta(end-numClassifiers+1:end,k)
          beta_hat = sum(omega .* net.beta(end-numClassifiers+1:end,k));
      end
      if beta_hat < net.threshold
        beta_hat = net.threshold;
      end
      net.w(net.t,k) = log(1/beta_hat);
    end
  end

```

```
89    % STEP 7: classifier voting weights
90    net.classifierweights{end+1} = net.w(end,:);
91
92    %%% Not great, should actually get.
93    f_measure = 0;
94    g_mean = 0;
95    recall = 0;
96    precision = 0;
97    err = 0;
98
99    net.initialized = 1;
100   net.t = net.t + 1;
101   netOut = net;
```

C.10 mse_reg.m

```
function mse_out = mse_reg(actual,label)
mse_out = (actual - label).^2;
end
```

C.11 regress_ensemble.m

```
function [predictions,posterior] = regress_ensemble(net, data, labels)
%%% TODO: MAKE INTO REGRESSION.
n_experts = length(net.classifiers);
weights = net.w(end,:);
if n_experts ~= length(weights)
  error('What are there are different number of weights and experts!')
end
p = zeros(n_experts, size(data,2));
normedWeights = weights/sum(weights);
predictions = zeros(1,size(data,2));
for k = 1:n_experts
  p(k,:) = net.classifiers{k}(data);
  predictions = predictions + p(k,:) * normedWeights(k);
end

posterior = 0; %%% FIX %p./repmat(sum(p,2),1,net.mclass);
```

C.12 rlm_update.m

```
function [S_new,P_new,w_new] = rlm_update(grad,p, P_old,w_old,t, alpha,err)
    % 0.97 < alpha < 1.
    % Error is received results compared to forward Prop'd results. Can be
    % done on a sample by sample basis, or on blocks.
    % Gradient would probably be current error minus last error.

    % Calculate Omega
    nGrad = length(grad.');
    omegaRow = zeros(nGrad,1);
    omegaRow(mod(t,length(grad.'))+1) = 1;

    Omega = [grad,omegaRow];

    % Calculate Lambda
    invLambda = [1,0;0,p];
    Lambda = pinv(invLambda);

    % Calculate new weights and intermediate matrices.
    S_new = alpha * Lambda + Omega.'*P_old*Omega;
    P_new = 1/alpha * (P_old - P_old*Omega*pinv(S_new)*Omega.'*P_old);
    w_new = w_old + P_new*grad*err;
end
```

C.13 trainrlm.m

```
%% trainrlm:
% Takes a MATLAB network class and a struct containing the different
% Parameters required to do iteration.
function [net,recurseOut,err_out] = trainrlm(net_old,recurseIn,X,Y,Xtest,Ytest)

    % X and Y can either be a single input-output combo,
    % or a batch of in-out combos.


    % Make sample-by-sample, batch by batch mode.
    repeat = 1;
    % Pass sample through NN.
    results_NN = net_old(Xtest);
    % Get Error.
    err = perform(net_old,Ytest,results_NN);%results_NN - Y;
    old_wb = getwb(net_old);
        while repeat == 1

            net_tmp = setwb(net_old,old_wb);
            time_tmp = recurseIn.Time;
            rho_tmp = recurseIn.rho;
            new_P = recurseIn.P;
            new_S = recurseIn.S;
            new_wb = old_wb;

            for i = 1:size(X,2)
                grad = defaultderiv('dperf_dwb',net_tmp, X(:,i),Y(:,i));%defaultderiv('dperf_dwb
    ',net_tmp, X(:,i),Y(:,i));
                [new_S, new_P,new_wb] =  rlm_update(grad,rho_tmp, new_P,new_wb,...
                                                    time_tmp, recurseIn.alpha,err);
                net_tmp = setwb(net_old, new_wb);
                time_tmp = time_tmp + 1;

            end
            results2_NN = net_tmp(Xtest);
            err_out = perform(net_tmp,Ytest,results2_NN);

            if err_out - err >= 1e-13
              % If the update increases error, don't actually update.
                recurseOut.rho = recurseIn.rho * 5;%10;
                if recurseOut.rho > 1e11
                    recurseOut.rho = 1e11;
                end
                    net = net_old;
                    repeat = 0;
            else%if err_out < 0.01
                net = net_tmp;
                recurseOut.S = new_S;
                recurseOut.P = new_P;
                recurseOut.rho = recurseIn.rho /5;%/ 10;
                if recurseOut.rho < 1e-30
                    recurseOut.rho = 1e-30;
                end
                repeat = 0;
            end
        end
    recurseOut.Time = time_tmp;
    new_rho = recurseOut.rho;
    recurseOut = recurseIn.updateMatrix(grad,time_tmp,new_P,new_S,new_rho,recurseIn.alpha);
end
```

Appendix D

C++ Code

D.1 RLNNCognitiveEngineTester_V3.cpp

```
#include <mlpack/core.hpp>
#include </home/tim/Desktop/rlnn5/RLNNCognitiveEngine.cpp>
#include </home/tim/Desktop/rlnn5/UDPServer.cpp>
#include </home/tim/Desktop/rlnn5/ViaSatDriver.cpp>
#include </home/tim/Desktop/rlnn5/EthUDPTransmitterSimple.cpp>
#include </home/tim/Desktop/rlnn5/ML605Driver.cpp>
#include </home/tim/Desktop/rlnn5/Logging.cpp>
//#include </home/tim/Desktop/rlnn5/ASRPDriver.cpp>
#include <boost/date_time/posix_time/posix_time.hpp>

#include <fstream>

#include <chrono>
#include <thread>

#include <iostream>

using namespace mlpack;
using namespace std::chrono;

int findIdxInActionList(const std::vector<int> &idxs, arma::Mat<int> & actionListIdxs) {
  int actionID = -1;
  bool skipFlag = false;

  for(int i=0; i<actionListIdxs.n_cols; i++) {
    skipFlag = false;
    for(int j=0; j<idxs.size() && skipFlag==false; j++) {
      if(actionListIdxs(j,i) != idxs[j]) {
        skipFlag = true;
      } else {
        //keep going
      }
    }
    if(skipFlag==false) {
      actionID = i;
      break;
    }
  }
  return actionID;
}

void doNothing(const boost::system::error_code&) {};


//-------------------------------------------------------//
//Main Function
//-------------------------------------------------------//
int main(int argc, char* argv[]) {

  //-------------------------------------------------------//
  //Command Line Inputs
  //-------------------------------------------------------//
  if(argc < 10) {
    std::cerr << "Usage: " << argv[0] << " fitnessWeights nParallelExploitNets
     nParallelExploreNets trainingBufferSize runTimeSec}" << std::endl
    << "fitnessWeights: [emergency,cooperation,balanced,powersaving,multimedia,launch]" << std::
     endl
    << "nParallelExploitNets: int[1,inf]" << std::endl
    << "nParallelExploreNets: int[1,inf]" << std::endl
    << "trainingBufferSize: int[1,inf]" << std::endl
    << "runTimeSec: int[0,inf]" << std::endl
```

```
60      << "Continue from File: int[0,1]"<<std::endl
61      << "FileName for loading: string"<<std::endl
62      << "FileName for saving: string"<<std::endl
63      << "SNR Profile to use: string"<<std::endl;
64      return 1;
65    }
66
67    arma::rowvec fitnessWeights;
68    std::string fileName,saveFileName;
69    int nParallelExploitNets;
70    int nParallelExploreNets;
71    int trainingBufferSize;
72    long runTimeSec;
73    int contFromFile;
74    std::string snrProfileUse;
75
76    if(std::string(argv[1])=="emergency") fitnessWeights = {0.1, 0.8, 0.025, 0.025, 0.025, 0.025};
77    else if(std::string(argv[1])=="cooperation") fitnessWeights =  {0.05, 0.05, 0.4, 0.4, 0.05,
        0.05};
78    else if(std::string(argv[1])=="balanced") fitnessWeights = {1.0/6.0, 1.0/6.0, 1.0/6.0,
        1.0/6.0, 1.0/6.0, 1.0/6.0};
79    else if(std::string(argv[1])=="powersaving") fitnessWeights = {0.05, 0.05, 0.05, 0.05, 0.3,
        0.5};
80    else if(std::string(argv[1])=="multimedia") fitnessWeights = {0.5, 0.3, 0.05, 0.05, 0.05,
        0.05};
81    else if(std::string(argv[1])=="launch") fitnessWeights = {0.2, 0.4, 0.1, 0.1, 0.1, 0.1};
82    else {std::cerr << "not a valid fitnessWeight name" << std::endl; return 1;}
83    nParallelExploitNets = std::stoi(argv[2]);
84    nParallelExploreNets = std::stoi(argv[3]);
85    trainingBufferSize = std::stoi(argv[4]);
86    runTimeSec = std::stoi(argv[5]);
87    contFromFile=std::stoi(argv[6]);
88    fileName = argv[7];
89    saveFileName = argv[8];
90    snrProfileUse = std::string(argv[9]);
91    //------------------------------------------------------//
92    //Simulation Parameters
93    //------------------------------------------------------//
94    boost::posix_time::ptime simStartTime;
95    simStartTime = boost::posix_time::microsec_clock::local_time();
96
97   #ifdef LOGGING
98    logFile << simStartTime << std::endl;
99    logFile << "::Start of Log File" << std::endl;
100   logFile << "--------------------" << std::endl;
101   #endif
102
103   //vars
104   //const int RSSI_UDP_PORT = REMOVED;
105   //const int FRAME_UDP_PORT = REMOVED;
106
107   //const std::vector<unsigned char> ETHTX_DEST_MAC_ADDR = REMOVED;
108   //const std::vector<unsigned char> ETHTX_SRC_MAC_ADDR = REMOVED;
109   //const unsigned int ETHTX_UDP_SRC_PORT = REMOVED;
110   //const unsigned int ETHTX_UDP_DEST_PORT = REMOVED;
111   const long ETHTX_TX_INTERVAL_MSEC = 1000;
112
113   const bool continueFromFile = contFromFile;
114   const bool saveToFile = true;
115   const std::string rlnnLoadFilename = fileName;
116   const std::string rlnnSaveFilename = saveFileName;
117
118   const bool SIMULATION_FLAG = true;
119   const bool USE_SNR_PROFILE = true;
120   const bool TX_POWER_PATCH_EN = true;
121
122   int actionID;
123   int lastActionID;
124   arma::mat actionList;
125   arma::Mat<int> actionListIdxs;
126   std::vector<int> actionIDElements;
127   actionIDElements.resize(4);
128   std::vector<double> _snrProfileVec;
129
130   //------------------------------------------------------//
131   //Setting up Params
132   //------------------------------------------------------//
133   //input parameters
134   std::cout<<"Setting Cog Engine Parameters" << std::endl;
135
136   CogEngParams cogEngParams;
137   //RLNNCognitiveEngine Params
138   cogEngParams.cogeng_epsilonResetLim = 4e-3;
139   cogEngParams.cogeng_nnExploreMaxPerfThresh = 0.9;
140   cogEngParams.cogeng_nnRejectionRate = 0.95;
141   cogEngParams.cogeng_trainFrac = 0.9;
142   cogEngParams.cogeng_pruneFrac = 0.75;
143   cogEngParams.cogeng_fitnessWeights = fitnessWeights;
144   cogEngParams.cogeng_forceExploreThreshold = 0.95;
145
146   //NeuralNetworkPredictor Params
147   cogEngParams.nnExplore_nNets = nParallelExploreNets;
```

```
148   cogEngParams.nnExplore_inputVectorSize = 7;
149   cogEngParams.nnExplore_hiddenLayerSizes.push_back(7);
150   cogEngParams.nnExplore_hiddenLayerSizes.push_back(50);
151   cogEngParams.nnExplore_outputVectorSize = 1;
152   cogEngParams.nnExplore_rmsProp_stepSize = 0.01;
153   cogEngParams.nnExplore_rmsProp_alpha = 0.88;
154   cogEngParams.nnExplore_rmsProp_eps = 1e-8;
155   cogEngParams.nnExplore_rmsProp_maxEpochs = 100;
156   cogEngParams.nnExplore_rmsProp_tolerance = 1e-18;
157   cogEngParams.nnExplore_rmsProp_shuffle = true;
158   cogEngParams.nnExploit_nNets = nParallelExploitNets;
159   cogEngParams.nnExploit_inputVectorSize = 7;
160   cogEngParams.nnExploit_hiddenLayerSizes.push_back(20);
161   cogEngParams.nnExploit_outputVectorSize = 1;
162   cogEngParams.nnExploit_rmsProp_stepSize = 0.01;
163   cogEngParams.nnExploit_rmsProp_alpha = 0.88;
164   cogEngParams.nnExploit_rmsProp_eps = 1e-8;
165   cogEngParams.nnExploit_rmsProp_maxEpochs = 100;
166   cogEngParams.nnExploit_rmsProp_tolerance = 1e-18;
167   cogEngParams.nnExploit_rmsProp_shuffle = true;
168
169   //Application Specific Object Params
170   cogEngParams.nnAppSpec_nOutVecFeatures = 8;
171   cogEngParams.nnAppSpec_frameSize = 16200.0;
172   cogEngParams.nnAppSpec_maxEsN0 = 40.0;//12.93; //dB
173   cogEngParams.nnAppSpec_minEsN0 = -20.0;//12.93; //dB
174   cogEngParams.nnAppSpec_maxBER = pow(10,-6);
175   cogEngParams.nnAppSpec_modList <<4<<4<<4<<4<<4<<4<<4<<4<<4<<4
176                   <<4<<8<<8<<8<<8<<8
177                   <<16<<16<<16<<16<<16
178                   <<32<<32<<32<<32
179                   <<arma::endr;
180   cogEngParams.nnAppSpec_codList  <<(1.0/4.0)<<(1.0/3.0)<<(2.0/5.0)<<(1.0/2.0)
181                    <<(3.0/5.0)<<(2.0/3.0)<<(3.0/4.0)<<(4.0/5.0)
182                    <<(5.0/6.0)<<(8.0/9.0)
183                    <<(3.0/5.0)<<(2.0/3.0)<<(3.0/4.0)<<(5.0/6.0)
184                    <<(8.0/9.0)
185                      <<(2.0/3.0)<<(3.0/4.0)<<(4.0/5.0)<<(5.0/6.0)
186                      <<(8.0/9.0)
187                      <<(3.0/4.0)<<(4.0/5.0)<<(5.0/6.0)<<(8.0/9.0)
188                     <<arma::endr;
189   cogEngParams.nnAppSpec_modCodList << 1 << 2 << 3 << 4 << 5 << 6 << 7 << 8 << 9 << 10
190                     << 12 << 13 << 14 << 15 << 16
191                     << 18 << 19 << 20 << 21 << 22
192                     << 24 << 25 << 26 << 27 << arma::endr;
193   cogEngParams.nnAppSpec_rollOffList << 0.35 << 0.25 << 0.20 <<arma::endr;
194   double RsMin = 0.5*pow(10,6)/(1+cogEngParams.nnAppSpec_rollOffList.min());//0.5
195   double RsMax = 5*pow(10,6)/(1+cogEngParams.nnAppSpec_rollOffList.max());
196   cogEngParams.nnAppSpec_symbolRateList <<(1.0*pow(10,6))<<arma::endr;
197   cogEngParams.nnAppSpec_transmitPowerList << -4.0
198                         << -3.5 << -3.0 << -2.5 << -2.0
199                         << -1.5 << -1.0 << -0.5 <<  0.0
200                         << arma::endr;
201   cogEngParams.buf_nTrainTestSamples = trainingBufferSize;//200;
202
203   #if NSE==1
204   cogEngParams.sigmoidSlope = 0.5;
205   cogEngParams.sigmoidThresh = 10;
206   cogEngParams.errorThresh = 10e-5;
207   #endif
208
209   #ifdef LOGGING
210   logFile << boost::posix_time::microsec_clock::local_time() << std::endl;
211   logFile << "::cogEngParams:" << std::endl;
212   logFile << "::cogEngParams.cogeng_epsilonResetLim: " << cogEngParams.cogeng_epsilonResetLim <<
        std::endl;
213   logFile << "::cogEngParams.cogeng_nnExploreMaxPerfThresh: " << cogEngParams.
       cogeng_nnExploreMaxPerfThresh << std::endl;
214   logFile << "::cogEngParams.cogeng_nnRejectionRate: " << cogEngParams.cogeng_nnRejectionRate <<
        std::endl;
215   logFile << "::cogEngParams.cogeng_fitnessWeights: " << cogEngParams.cogeng_fitnessWeights <<
       std::endl;
216   logFile << "::cogEngParams.cogeng_trainFrac: " << cogEngParams.cogeng_trainFrac << std::endl;
217   logFile << "::cogEngParams.cogeng_pruneFrac: " << cogEngParams.cogeng_pruneFrac << std::endl;
218   logFile << "::cogEngParams.cogeng_forceExploreThreshold: " << cogEngParams.
       cogeng_forceExploreThreshold << std::endl;
219
220   //NeuralNetworkPredictor Params
221   logFile << "::cogEngParams.nnExplore_nNets: " << cogEngParams.nnExplore_nNets << std::endl;
222   logFile << "::cogEngParams.nnExplore_inputVectorSize: " << cogEngParams.
       nnExplore_inputVectorSize << std::endl;
223   logFile << "::cogEngParams.nnExplore_hiddenLayerSizes: ";
224   for(int j=0; j<cogEngParams.nnExplore_hiddenLayerSizes.size(); j++) {
225     logFile << cogEngParams.nnExplore_hiddenLayerSizes[j] << '\t';
226   }
227   logFile << std::endl;
228   logFile << "::cogEngParams.nnExplore_outputVectorSize: " << cogEngParams.
       nnExplore_outputVectorSize << std::endl;
229   logFile << "::cogEngParams.nnExploit_nNets: " << cogEngParams.nnExploit_nNets << std::endl;
230   logFile << "::cogEngParams.nnExploit_inputVectorSize: " << cogEngParams.
       nnExploit_inputVectorSize << std::endl;
231   logFile << "::cogEngParams.nnExploit_hiddenLayerSizes: ";
```

```
232   for(int j=0; j<cogEngParams.nnExploit_hiddenLayerSizes.size(); j++) {
233     logFile << cogEngParams.nnExploit_hiddenLayerSizes[j] << '\t';
234   }
235   logFile << std::endl;
236   logFile << "::cogEngParams.nnExploit_outputVectorSize: " << cogEngParams.
        nnExploit_outputVectorSize << std::endl;
237
238   //RMSProp params
239   logFile << "::cogEngParams.nnExplore_rmsProp_stepSize: " << cogEngParams.
        nnExplore_rmsProp_stepSize << std::endl;
240   logFile << "::cogEngParams.nnExplore_rmsProp_alpha: " << cogEngParams.nnExplore_rmsProp_alpha
        << std::endl;
241   logFile << "::cogEngParams.nnExplore_rmsProp_eps: " << cogEngParams.nnExplore_rmsProp_eps <<
        std::endl;
242   logFile << "::cogEngParams.nnExplore_rmsProp_maxEpochs: " << cogEngParams.
        nnExplore_rmsProp_maxEpochs << std::endl;
243   logFile << "::cogEngParams.nnExplore_rmsProp_tolerance: " << cogEngParams.
        nnExplore_rmsProp_tolerance << std::endl;
244   logFile << "::cogEngParams.nnExplore_rmsProp_shuffle: " << cogEngParams.
        nnExplore_rmsProp_shuffle << std::endl;
245   logFile << "::cogEngParams.nnExploit_rmsProp_stepSize: " << cogEngParams.
        nnExploit_rmsProp_stepSize << std::endl;
246   logFile << "::cogEngParams.nnExploit_rmsProp_alpha: " << cogEngParams.nnExploit_rmsProp_alpha
        << std::endl;
247   logFile << "::cogEngParams.nnExploit_rmsProp_eps: " << cogEngParams.nnExploit_rmsProp_eps <<
        std::endl;
248   logFile << "::cogEngParams.nnExploit_rmsProp_maxEpochs: " << cogEngParams.
        nnExploit_rmsProp_maxEpochs << std::endl;
249   logFile << "::cogEngParams.nnExploit_rmsProp_tolerance: " << cogEngParams.
        nnExploit_rmsProp_tolerance << std::endl;
250   logFile << "::cogEngParams.nnExploit_rmsProp_shuffle: " << cogEngParams.
        nnExploit_rmsProp_shuffle << std::endl;
251
252   //App Specific Params
253   logFile << "::cogEngParams.nnAppSpec_nOutVecFeatures: " << cogEngParams.
        nnAppSpec_nOutVecFeatures << std::endl;
254   logFile << "::cogEngParams.nnAppSpec_frameSize: " << cogEngParams.nnAppSpec_frameSize << std::
        endl;
255   logFile << "::cogEngParams.nnAppSpec_maxEsN0: " << cogEngParams.nnAppSpec_maxEsN0 << std::endl
        ;
256   logFile << "::cogEngParams.nnAppSpec_maxBER: " << cogEngParams.nnAppSpec_maxBER << std::endl;
257   logFile << "::cogEngParams.nnAppSpec_modList: " << cogEngParams.nnAppSpec_modList << std::endl
        ;
258   logFile << "::cogEngParams.nnAppSpec_codList: " << cogEngParams.nnAppSpec_codList << std::endl
        ;
259   logFile << "::cogEngParams.nnAppSpec_rollOffList: " << cogEngParams.nnAppSpec_rollOffList <<
        std::endl;
260   logFile << "::cogEngParams.nnAppSpec_symbolRateList: " << cogEngParams.
        nnAppSpec_symbolRateList << std::endl;
261   logFile << "::cogEngParams.nnAppSpec_transmitPowerList: " << cogEngParams.
        nnAppSpec_transmitPowerList << std::endl;
262   logFile << "::cogEngParams.nnAppSpec_modCodList: " << cogEngParams.nnAppSpec_modCodList << std
        ::endl;
263
264   //TrainingDataBuffer Params
265   logFile << "::cogEngParams.buf_nTrainTestSamples: " << cogEngParams.buf_nTrainTestSamples <<
        std::endl;
266   logFile << "------------------" << std::endl;
267
268   logFile << "::RLNNTester Params:" << std::endl;
269   logFile << "::continueFromFile: " <<  continueFromFile << std::endl;
270   logFile << "::saveToFile: " << saveToFile << std::endl;
271   logFile << "::rlnnLoadFilename: " << rlnnLoadFilename << std::endl;
272   logFile << "::rlnnSaveFilename: " << rlnnSaveFilename << std::endl;
273   logFile << "::SIMULATION_FLAG: " << SIMULATION_FLAG << std::endl;
274   logFile << "::TX_POWER_PATCH_EN: " << TX_POWER_PATCH_EN << std::endl;
275
276   #endif
277
278   actionList.set_size(6,cogEngParams.nnAppSpec_symbolRateList.n_elem
279               *cogEngParams.nnAppSpec_transmitPowerList.n_elem
280               *cogEngParams.nnAppSpec_modList.n_elem
281               *cogEngParams.nnAppSpec_rollOffList.n_elem);
282   actionListIdxs.set_size(4,cogEngParams.nnAppSpec_symbolRateList.n_elem
283               *cogEngParams.nnAppSpec_transmitPowerList.n_elem
284               *cogEngParams.nnAppSpec_modList.n_elem
285               *cogEngParams.nnAppSpec_rollOffList.n_elem);
286   int id = 0;
287   for(int i1=0; i1<cogEngParams.nnAppSpec_symbolRateList.n_elem; i1++) {
288     for(int i2=0; i2<cogEngParams.nnAppSpec_transmitPowerList.n_elem; i2++) {
289       for(int i3=0; i3<cogEngParams.nnAppSpec_modList.n_elem; i3++) {
290         for(int i4=0; i4<cogEngParams.nnAppSpec_rollOffList.n_elem; i4++) {
291             actionList(0,id) = cogEngParams.nnAppSpec_symbolRateList(i1);
292             actionListIdxs(0,id) = i1;
293             actionList(1,id) = cogEngParams.nnAppSpec_transmitPowerList(i2);
294             actionListIdxs(1,id) = i2;
295             actionList(2,id) = (double) cogEngParams.nnAppSpec_modCodList(i3); //modcod
296             actionListIdxs(2,id) = i3;
297             actionList(3,id) = cogEngParams.nnAppSpec_rollOffList(i4);
298             actionListIdxs(3,id) = i4;
299             id++;
300         }
```

```
      }
    }
  }

  for(int i=0; i<actionList.n_cols; i++) {
    actionList(4,i) = (double) cogEngParams.nnAppSpec_modList(actionListIdxs(2,i));
    actionList(5,i) = cogEngParams.nnAppSpec_codList(actionListIdxs(2,i));
    actionList(2,i) = log2(actionList(4,i));
  }

  //Save all actions in log
  #ifdef LOGGING
  logFile << boost::posix_time::microsec_clock::local_time() << std::endl;
  logFile << "::Action List:" << std::endl;
  for(int i=0; i<actionList.n_cols; i++) {
    for(int j=0; j<actionList.n_rows; j++) {
      logFile << actionList(j,i) << "\t";
    }
    logFile << std::endl;
  }
  logFile << "--------------------" << std::endl;
  #endif

  // save all action indices in log
  #ifdef LOGGING
  logFile << boost::posix_time::microsec_clock::local_time() << std::endl;
  logFile << "::Action Idxs:" << std::endl;
  for(int i=0; i<actionListIdxs.n_cols; i++) {
    for(int j=0; j<actionListIdxs.n_rows; j++) {
      logFile << actionListIdxs(j,i) << "\t";
    }
    logFile << std::endl;
  }
  logFile << "--------------------" << std::endl;
  #endif


  cogEngParams.buf_actionList = actionList;

  //-------------------------------------------------------//
  //Instantiations and Set-up
  //-------------------------------------------------------//
  //instantiate cog engine
  std::cout<<"Instantiating Cog Engine" << std::endl;
  RLNNCognitiveEngine rlnnCogEng(cogEngParams);
  if(continueFromFile) {
    rlnnCogEng.loadOldRun(rlnnLoadFilename);
  }

  //instantiate UDP TLM Receiver
  ViaSatDriver vsDrvr;
  ViaSatDriver frameDrvr;
  ViaSatDriver::RSSIMessageFormat rssiMsg;
  ViaSatDriver::FrameMessageFormat frameMsg;
  int udp_status;

  //instantiate UDP transmitter
  ML605Driver ml605;
  ML605Driver::ML605MessageFormat ml605Msg;
  std::vector<unsigned char> ml605Buf;

  // ASRPDriver asrp("IP ADDRESS:PORT, REMOVED");
  //init socket to listen for EsN0/RSSI on port 13
  boost::asio::io_service io_service;
  // boost::asio::io_service::work work(io_service);

  UDPServer rssiServer(io_service,RSSI_UDP_PORT);
  UDPServer frameServer(io_service,FRAME_UDP_PORT);
  std::vector<unsigned char> * rssiBuff;
  std::vector<unsigned char> * frameBuff;
  int packetSize;

  //init socket to send to ML605
  EthUDPTransmitterSimple ethTx(io_service, ETHTX_UDP_DEST_PORT, ETHTX_UDP_SRC_PORT,
      ETHTX_DEST_MAC_ADDR, ETHTX_SRC_MAC_ADDR);

  //open sockets and continuously listen
  std::thread tRSSI([&io_service](){ io_service.run(); });
  std::thread tFrames([&io_service](){ io_service.run(); });

  //set up way to stop execution cleanly
  bool quitExecution = false;

  //load in SNR Profile (if enabled)
  if(USE_SNR_PROFILE) {
    std::string line;
    std::string snrFileLoc;

    if (snrProfileUse == "Excellent"){
      snrFileLoc = "snrFiles/revert_norm_Peregrin_log_160426_163653_esno.txt";
      //std::cout <<"::: SNR PROFILE EXCELLENT";
    }
```

```
393       else if(snrProfileUse == "Great"){
394         snrFileLoc = "snrFiles/revert_norm_Peregrin_log_160429_104439_esno.txt"; //"Great" pass
395       }
396       else if(snrProfileUse == "Good"){
397         snrFileLoc = "snrFiles/revert_norm_Peregrin_log_160510_122415_esno.txt"; //"Good" pass
398       }
399       else if(snrProfileUse == "Bad"){
400         snrFileLoc = "snrFiles/revert_norm_Peregrin_log_160419_161748_esno.txt"; //"Good" pass
401       }
402       else{
403         snrFileLoc = "snrProfile.txt";
404       }
405
406     std::ifstream snrProfileFile(snrFileLoc); //"Excellent" pass
407
408
409       while(std::getline(snrProfileFile,line)) {
410         std::istringstream ss(line);
411         std::string token;
412
413         while(std::getline(ss,token)) {
414           _snrProfileVec.push_back(std::atof(token.c_str()));
415         }
416       }
417       snrProfileFile.close();
418     }
419
420     //-------------------------------------------------------//
421     //Main Execution
422     //-------------------------------------------------------//
423     //choose an action
424     arma::rowvec measurementVec(6);
425     double rcvdEsN0;
426     int i=0;
427     for(unsigned int i=0; /*(i<20000) &&*/ (!quitExecution) ; i++) {
428
429       #ifdef LOGGING
430       logFile << "::Iteration: " << i << std::endl;
431       logFile << "::Start Time: " << boost::posix_time::microsec_clock::local_time() << std::endl;
432       #endif
433
434       //ACTION CHOOSING------------------------------//
435       //choose action
436       std::cout<<i<<": Choosing Action"<<std::endl;
437       actionID = rlnnCogEng.chooseAction();
438
439       if(TX_POWER_PATCH_EN) {
440         actionIDElements[0] = actionListIdxs(0,actionID);
441         actionIDElements[2] = actionListIdxs(2,actionID);
442         actionIDElements[3] = actionListIdxs(3,actionID);
443         if(i != 0) {
444           if(std::abs(actionList(1,actionID)-actionList(1,lastActionID)) > 1.5) { //if tx pwr
     change is more than 1.5 dB
445             if((actionList(1,actionID)-actionList(1,lastActionID)) > 1.5) { //new power is >1.5 dB
      larger
446               //find action with pwr 1.5 dB larger
447               actionIDElements[1] = actionListIdxs(1,lastActionID)+3; //each idx is 0.5 dB steps.
     3*0.5 dB = 1.5 dB
448               actionID = findIdxInActionList(actionIDElements,actionListIdxs);
449
450             } else { //new power is >1.5 dB smaller
451               //find action with pwr 1.5 dB smaller
452               actionIDElements[1] = actionListIdxs(1,lastActionID)-3; //each idx is 0.5 dB steps.
     3*0.5 dB = 1.5 dB
453               actionID = findIdxInActionList(actionIDElements,actionListIdxs);
454             }
455           }
456
457         } else {
458           if(std::abs(0.0 - actionList(1,actionID)) > 1.5) { //power starts at 0.00 dB by default
459             //find action with pwr 1.5 dB smaller
460             actionIDElements[1] = actionListIdxs.row(1).max()-3; //each idx is 0.5 dB steps. 3*0.5
      dB = 1.5 dB
461             actionID = findIdxInActionList(actionIDElements,actionListIdxs);
462           }
463         }
464         lastActionID = actionID;
465       }
466
467       std::cout<<i<<": Action Chosen: " << actionID << std::endl;
468       std::cout<<i<<": Symbol Rate Idx: " << actionListIdxs(0,actionID) << std::endl;
469       std::cout<<i<<": TX Power Idx: " << actionListIdxs(1,actionID) << std::endl;
470       std::cout<<i<<": ModCod Idx: " << actionListIdxs(2,actionID) << std::endl;
471       std::cout<<i<<": Roll Off Idx: " << actionListIdxs(3,actionID) << std::endl;
472       std::cout<<i<< ": ModCod: " << cogEngParams.nnAppSpec_modCodList(actionListIdxs(2,actionID))
       << std::endl;
473
474
475       #ifdef LOGGING
476       logFile << "::Action Chosen: " << boost::posix_time::microsec_clock::local_time() << std::
     endl;
477       logFile << "::Action ID: " << actionID << std::endl;
```

```
478       logFile << "::Action Params: ";
479       for(int i=0; i<actionList.n_rows; i++) {
480         logFile << actionList(i,actionID) << "\t";
481       }
482       logFile << std::endl;
483       #endif
484
485       //send action
486       std::cout<<i<<": Transmitting Action"<<std::endl;
487       ml605Msg.modcod = cogEngParams.nnAppSpec_modCodList(actionListIdxs(2,actionID));
488       ml605Msg.rolloff = actionListIdxs(3,actionID);
489       ml605Msg.transmitPower = actionListIdxs(1,actionID);
490       //ml605Msg.transmitPower = 15;
491       ml605Msg.enable = 1;
492       ml605.generateTXActionMessage(ml605Msg,ml605Buf);
493       ethTx.updateUDPPayload(ml605Buf);
494       ethTx.sendFrame();
495       // asrp.sendAction(cogEngParams.nnAppSpec_modCodList(actionListIdxs(2,actionID)), actionList
          (1,actionID), actionList(3,actionID));
496
497       //RECORDING RESPONSE-------------------------------//
498       //std::cout << "Send next UDP packet to be received." << std::endl;
499       #ifdef LOGGING
500       logFile << "::Action Sent: " << boost::posix_time::microsec_clock::local_time() << std::endl
          ;
501       #endif
502
503       //block until worst case RTT has passed
504       boost::asio::deadline_timer t(io_service,boost::posix_time::milliseconds(40));
505       t.wait();
506
507       #ifdef LOGGING
508       logFile << "::Action Received: " << boost::posix_time::microsec_clock::local_time() << std::
          endl;
509       #endif
510
511       if(!SIMULATION_FLAG) {
512         //read response from rssi packet
513         std::cout<<i<<": Reading Received Action" << std::endl;
514         rssiBuff = rssiServer.getRecvBuffer();
515         packetSize = rssiServer.getPacketSize();
516         udp_status = vsDrvr.getRSSIPacketContents(rssiBuff,rssiMsg,packetSize);
517         if(rssiMsg.rcvLock) {
518           rcvdEsN0 = rssiMsg.esN0;
519         }
520         std::cout<<i<<": Receive Lock: " << rssiMsg.rcvLock << std::endl;
521         #ifdef LOGGING
522           logFile << "::Receive Lock: " << rssiMsg.rcvLock << std::endl;
523         #endif
524         //read response from frames
525         //udp_status = 0;
526         //while(!udp_status) { //we don't want to block because frames might be missing
527           //frameBuff = frameServer.getRecvBuffer();
528           //packetSize = frameServer.getPacketSize();
529           //udp_status = frameDrvr.getFramePacketContents(frameBuff,frameMsg,packetSize);
530         //}
531         //populate measurement vector
532         measurementVec(0) = rcvdEsN0;
533         measurementVec(1) = cogEngParams.nnAppSpec_transmitPowerList(actionListIdxs(1,actionID));
          //tx power
534         measurementVec(2) = cogEngParams.nnAppSpec_symbolRateList(actionListIdxs(0,actionID)); //
          Rs
535         measurementVec(3) = cogEngParams.nnAppSpec_rollOffList(actionListIdxs(3,actionID)); //
          rolloff
536         measurementVec(4) = cogEngParams.nnAppSpec_modList(actionListIdxs(2,actionID)); //mod
537         measurementVec(5) = cogEngParams.nnAppSpec_codList(actionListIdxs(2,actionID)); //cod
538       } else {
539         double SNR;
540         if(USE_SNR_PROFILE) {
541           SNR = _snrProfileVec[i];
542         } else {
543           if(i<6000) {
544             SNR=((double)i)*0.002*2;
545           } else {
546             SNR=12.0-0.002*(i-6000)*2;
547           }
548         }
549         double noise = mlpack::math::RandNormal(0,0.01); //std deviation ~0.1 dB
550
551         measurementVec(0) = SNR + noise + cogEngParams.nnAppSpec_transmitPowerList(actionListIdxs
          (1,actionID));
552         measurementVec(1) = cogEngParams.nnAppSpec_transmitPowerList(actionListIdxs(1,actionID));
          //tx power
553         measurementVec(2) = cogEngParams.nnAppSpec_symbolRateList(actionListIdxs(0,actionID)); //
          Rs
554         measurementVec(3) = cogEngParams.nnAppSpec_rollOffList(actionListIdxs(3,actionID)); //
          rolloff
555         measurementVec(4) = cogEngParams.nnAppSpec_modList(actionListIdxs(2,actionID)); //mod
556         measurementVec(5) = cogEngParams.nnAppSpec_codList(actionListIdxs(2,actionID)); //cod
557       }
558       //we don't need to measure log2(M) since we have M.
559       std::cout <<i<<": measurementVec: " << std::endl;
```

```
560       std::cout << measurementVec << std::endl;
561
562       #ifdef LOGGING
563       logFile << "::Measurement Received:";
564       logFile << measurementVec;
565       #endif
566
567       //record response of environment
568       std::cout<<i<<": Recording Response" << std::endl;
569       rlnnCogEng.recordResponse(actionID, measurementVec);
570       #ifdef LOGGING
571       logFile << "::End Time: " << boost::posix_time::microsec_clock::local_time() << std::endl;
572       logFile << std::endl;
573       #endif
574
575       if((boost::posix_time::microsec_clock::local_time()-simStartTime).total_seconds() >
          runTimeSec) {
576         quitExecution = true;
577       }
578     }
579
580     rssiServer.close();
581     frameServer.close();
582     tRSSI.join();
583     tFrames.join();
584
585     logFile.close();
586     if(saveToFile) {
587       std::cout << rlnnSaveFilename;
588       rlnnCogEng.saveCurrentRun(rlnnSaveFilename);
589     }
590     return 0;
591   }
```

D.2 RLNNCognitiveEngine.cpp

```
#include <mlpack/core.hpp>

#include <mlpack/methods/ann/activation_functions/logistic_function.hpp>
#include <mlpack/methods/ann/activation_functions/identity_function.hpp>

#include <mlpack/methods/ann/init_rules/random_init.hpp>

#include <mlpack/methods/ann/layer/linear_layer.hpp>
#include <mlpack/methods/ann/layer/base_layer.hpp>
#include <mlpack/methods/ann/layer/identity_output_layer.hpp>

#include <mlpack/methods/ann/ffn.hpp>
#include <mlpack/methods/ann/performance_functions/mse_function.hpp>
#include <mlpack/core/optimizers/rmsprop/rmsprop.hpp>
#include <mlpack/core/optimizers/gradient_descent/gradient_descent.hpp>
#include <mlpack/core/optimizers/adadelta/ada_delta.hpp>
#include <mlpack/core/optimizers/sgd/sgd.hpp>

#include "/home/tim/Desktop/rlnn5/NeuralNetworkPredictor.cpp"
#include "/home/tim/Desktop/rlnn5/LearnNSEPredictor.cpp"
#include "/home/tim/Desktop/rlnn5/TrainingDataBuffer.cpp"
#include "/home/tim/Desktop/rlnn5/Logging.hpp"
#include "/home/tim/Desktop/rlnn5/MemoryManagement.hpp"
#include "/home/tim/Desktop/rlnn5/ApplicationSpecificHelper.cpp"

#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <mlpack/prereqs.hpp>
#include <fstream>
#include <boost/archive/tmpdir.hpp>
#include <boost/serialization/base_object.hpp>
#include <boost/serialization/utility.hpp>
#include <boost/serialization/list.hpp>
#include <boost/serialization/assume_abstract.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/serialization/serialization.hpp>
#include <thread>
#include <iostream>
#include <condition_variable>
#include <boost/date_time/posix_time/posix_time.hpp>
#include <chrono>

using namespace mlpack;

struct CogEngParams {
  //RLNNCognitiveEngine Params
  double cogeng_epsilonResetLim;
  double cogeng_nnExploreMaxPerfThresh;
  double cogeng_nnRejectionRate;
  double cogeng_trainFrac;
  double cogeng_pruneFrac;
  arma::rowvec cogeng_fitnessWeights;
  double cogeng_forceExploreThreshold;
  //NeuralNetworkPredictorExplore Params
  int nnExplore_nNets;
  int nnExplore_inputVectorSize;
  std::vector<int> nnExplore_hiddenLayerSizes;
  int nnExplore_outputVectorSize;
  double nnExplore_rmsProp_stepSize;
  double nnExplore_rmsProp_alpha;
  double nnExplore_rmsProp_eps;
  size_t nnExplore_rmsProp_maxEpochs;
  double nnExplore_rmsProp_tolerance;
  bool nnExplore_rmsProp_shuffle;
  //NeuralNetworkPredictor Params
  int nnExploit_nNets;
  int nnExploit_inputVectorSize;
  std::vector<int> nnExploit_hiddenLayerSizes;
  int nnExploit_outputVectorSize;
  double nnExploit_rmsProp_stepSize;
  double nnExploit_rmsProp_alpha;
  double nnExploit_rmsProp_eps;
  size_t nnExploit_rmsProp_maxEpochs;
  double nnExploit_rmsProp_tolerance;
  bool nnExploit_rmsProp_shuffle;
  //Application Specific Object Params
  int nnAppSpec_nOutVecFeatures;
  double nnAppSpec_frameSize;
  double nnAppSpec_maxEsN0;
  double nnAppSpec_minEsN0;
  double nnAppSpec_maxBER;
  arma::Row<int> nnAppSpec_modList;
  arma::Row<double> nnAppSpec_codList;
  arma::Row<double> nnAppSpec_rollOffList;
  arma::Row<double> nnAppSpec_symbolRateList;
  arma::Row<double> nnAppSpec_transmitPowerList;
  arma::Row<int> nnAppSpec_modCodList;
  //Learn NSE parameters
```

```
  #if NSE==1
    double sigmoidSlope;
    double sigmoidThresh;
    double errorThresh;
  #endif
  //TrainingDataBuffer Params
  int buf_nTrainTestSamples;
  arma::mat buf_actionList;
};

namespace boost {
namespace serialization {
class access;
}
}

class RLNNCognitiveEngine {
  friend std::ostream & operator<<(std::ostream &os, const RLNNCognitiveEngine &rlnnCogEng);
  public:
    #if NSE==1
      LearnNSEPredictor<ThreeLayerNetwork,optimization::RMSprop> nnExplore;
      LearnNSEPredictor<ThreeLayerNetwork,optimization::RMSprop> nnExploreTrainer;

      std::vector<LearnNSEPredictor<TwoLayerNetwork,optimization::RMSprop> *> nnExploit;
      std::vector<LearnNSEPredictor<TwoLayerNetwork,optimization::RMSprop> *> nnExploitTrainer;
    #elif LM==1
      NeuralNetworkPredictor<ThreeLayerNetwork,optimization::RMSprop> nnExplore;
      NeuralNetworkPredictor<ThreeLayerNetwork,optimization::RMSprop> nnExploreTrainer;

      std::vector<NeuralNetworkPredictor<TwoLayerNetwork,optimization::RMSprop> *> nnExploit;
      std::vector<NeuralNetworkPredictor<TwoLayerNetwork,optimization::RMSprop> *>
     nnExploitTrainer;
    #elif RLM==1
      NeuralNetworkPredictor<ThreeLayerNetwork,optimization::RMSprop> nnExplore;
      NeuralNetworkPredictor<ThreeLayerNetwork,optimization::RMSprop> nnExploreTrainer;

      std::vector<NeuralNetworkPredictor<TwoLayerNetwork,optimization::RMSprop> *> nnExploit;
      std::vector<NeuralNetworkPredictor<TwoLayerNetwork,optimization::RMSprop> *>
     nnExploitTrainer;
    #endif



    TrainingDataBuffer trBuf;
    ApplicationSpecificHelper appSpecObj;

    RLNNCognitiveEngine(const CogEngParams & inpParams);
    RLNNCognitiveEngine();
    int chooseAction();
    void recordResponse(int actionID, const arma::rowvec &measurementVec);

    void saveCurrentRun(std::string filename) {
      //save nn weights
      nnExplore.saveCurrentRun(filename + "_nnExplore" + ".txt");
      for(int i=0; i<nnExploit.size(); i++) {
        nnExploit[i]->saveCurrentRun(filename + "_nnExploit_" + std::to_string(i) + ".txt");
      }
      //save training buffers
      trBuf.saveCurrentRun(filename + "_trBuf" + ".txt");
      //save app specific vars
      appSpecObj.saveCurrentRun(filename + "_appSpecObj" + ".txt");

      std::ofstream ofs(filename + "_RLNNCogEng" ".txt");
      //save data to archive
      boost::archive::text_oarchive oa(ofs);
      //write class instance to archive
      oa & *this;
      //archive and stream closed when destructors are called
    };

    void loadOldRun(std::string filename) {
      std::ifstream ifs(filename + "_RLNNCogEng" ".txt");
      boost::archive::text_iarchive ia(ifs);
      //read class state from archive
      ia & *this;
      //archive and stream closed when destructors are called.

      //load nn weights
      nnExplore.loadOldRun(filename + "_nnExplore" + ".txt");
      for(int i=0; i<nnExploit.size(); i++) {
        nnExploit[i]->loadOldRun(filename + "_nnExploit_" + std::to_string(i) + ".txt");
      }
      //load training buffers
      trBuf.loadOldRun(filename + "_trBuf" + ".txt");
      //load app specific vars
      appSpecObj.loadOldRun(filename + "_appSpecObj" + ".txt");
    }

  private:
    double _trainFrac;
    double _pruneFrac;
    double _nnRejectionRate;
```

```
179     double _epsilon;
180     int _epsilonIter;
181     double _epsilonResetLim;
182     double _nnExploreMaxPerfThresh;
183     bool _nnTrained;
184     arma::mat _actionList;
185     int _nActions;
186     bool _exploitFlag;
187     bool _forceExplore;
188     bool _firstExploreAfterNNTrained;
189     arma::rowvec _fitnessWeights;
190     double _fitObservedMax;
191     double _lastExploitFitObserved;
192     arma::mat _nnExploreInputs;
193     arma::mat _nnExploitInputs;
194     int _histRollBackIdx;
195     int _histRollBackCnt;
196 
197     bool _currentlyTraining;
198     std::atomic<bool> _trainingComplete;
199 
200     std::vector<std::thread> tTrainingVec;
201 
202     double _forceExploreThreshold;
203 
204     friend class boost::serialization::access;
205 
206     template<class Archive>
207     void serialize(Archive & ar, const int version) {
208       ar & _epsilon;
209       ar & _epsilonIter;
210       ar & _nnTrained;
211       ar & _firstExploreAfterNNTrained;
212       ar & _fitObservedMax;
213       ar & _lastExploitFitObserved;
214       ar & _nnExploreInputs;
215       ar & _nnExploitInputs;
216       ar & _exploitFlag;
217       ar & _forceExplore;
218       ar & _histRollBackIdx;
219     }
220 };
221 
222 BOOST_SERIALIZATION_ASSUME_ABSTRACT(RLNNCognitiveEngine)
223 
224 // Initialization
225 RLNNCognitiveEngine::RLNNCognitiveEngine(const CogEngParams & inpParams)
226 #if NSE==1
227 :    nnExplore(inpParams.nnExplore_nNets,inpParams.nnExplore_inputVectorSize,inpParams.
        nnExplore_hiddenLayerSizes,inpParams.nnExplore_outputVectorSize,inpParams.sigmoidSlope,
        inpParams.sigmoidThresh,inpParams.errorThresh),
228   nnExploreTrainer(inpParams.nnExplore_nNets,inpParams.nnExplore_inputVectorSize,inpParams.
        nnExplore_hiddenLayerSizes,inpParams.nnExplore_outputVectorSize,inpParams.sigmoidSlope,
        inpParams.sigmoidThresh,inpParams.errorThresh),
229 #else
230 :    nnExplore(inpParams.nnExplore_nNets,inpParams.nnExplore_inputVectorSize,inpParams.
        nnExplore_hiddenLayerSizes,inpParams.nnExplore_outputVectorSize),
231   nnExploreTrainer(inpParams.nnExplore_nNets,inpParams.nnExplore_inputVectorSize,inpParams.
        nnExplore_hiddenLayerSizes,inpParams.nnExplore_outputVectorSize),
232 #endif
233     trBuf(inpParams.nnAppSpec_nOutVecFeatures,inpParams.buf_nTrainTestSamples,2), //2 NNs (
        explore and exploit)
234     appSpecObj( inpParams.buf_actionList,
235           inpParams.nnAppSpec_frameSize,
236             inpParams.nnAppSpec_maxEsN0,
237             inpParams.nnAppSpec_minEsN0,
238             inpParams.nnAppSpec_modList,
239             inpParams.nnAppSpec_codList,
240             inpParams.nnAppSpec_rollOffList,
241             inpParams.nnAppSpec_symbolRateList,
242             inpParams.nnAppSpec_transmitPowerList,
243             inpParams.nnAppSpec_modCodList,
244             inpParams.nnAppSpec_maxBER)
245 
246 
247 {
248 
249   //RLNNCognitiveEngine
250   _epsilonResetLim = inpParams.cogeng_epsilonResetLim;
251   _nnExploreMaxPerfThresh = inpParams.cogeng_nnExploreMaxPerfThresh;
252   _nnRejectionRate = inpParams.cogeng_nnRejectionRate;
253   _trainFrac = inpParams.cogeng_trainFrac;
254   _pruneFrac = inpParams.cogeng_pruneFrac;
255   _forceExploreThreshold = inpParams.cogeng_forceExploreThreshold;
256 
257   _epsilon = 1.0;
258   _epsilonIter = 1;
259   _nnTrained = false;
260   _firstExploreAfterNNTrained = false;
261   _actionList = inpParams.buf_actionList;
262   _nActions = _actionList.n_cols;
263   _exploitFlag = false;
```

```
264   _fitnessWeights = inpParams.cogeng_fitnessWeights;
265   _fitObservedMax = 0.0;
266   _lastExploitFitObserved = 0.0;
267   _forceExplore = true;
268   _histRollBackIdx = -1;
269   _histRollBackCnt = 0;
270
271   _currentlyTraining = false;
272   _trainingComplete = false;
273
274   //NeuralNetworkPredictors
275   //configure RMS prop for nnExplore
276   /*******TODO: Check if RMSProp is actually used at any point******/
277   nnExplore.initializeRMSProp(inpParams.nnExplore_rmsProp_stepSize,
278                 inpParams.nnExplore_rmsProp_alpha,
279                 inpParams.nnExplore_rmsProp_eps,
280                 inpParams.nnExplore_rmsProp_maxEpochs*inpParams.buf_nTrainTestSamples,
281                 inpParams.nnExplore_rmsProp_tolerance,
282                 inpParams.nnExplore_rmsProp_shuffle
283                 );
284   nnExploreTrainer.initializeRMSProp(inpParams.nnExplore_rmsProp_stepSize,
285                 inpParams.nnExplore_rmsProp_alpha,
286                 inpParams.nnExplore_rmsProp_eps,
287                 inpParams.nnExplore_rmsProp_maxEpochs*inpParams.buf_nTrainTestSamples,
288                 inpParams.nnExplore_rmsProp_tolerance,
289                 inpParams.nnExplore_rmsProp_shuffle
290                 );
291   //generate array of NNs for nnExploit (one for each type of action)
292   for(int i=0; i<_actionList.n_rows; i++) {
293     #if NSE==1
294       LearnNSEPredictor<TwoLayerNetwork,optimization::RMSprop> * pNNExploit =
295         new LearnNSEPredictor<TwoLayerNetwork,optimization::RMSprop>(
296           inpParams.nnExploit_nNets,
297           inpParams.nnExploit_inputVectorSize,
298           inpParams.nnExploit_hiddenLayerSizes,
299           inpParams.nnExploit_outputVectorSize,
300           inpParams.sigmoidSlope,
301           inpParams.sigmoidThresh,
302           inpParams.errorThresh
303           );
304     #else
305       NeuralNetworkPredictor<TwoLayerNetwork,optimization::RMSprop> * pNNExploit =
306         new NeuralNetworkPredictor<TwoLayerNetwork,optimization::RMSprop>(
307           inpParams.nnExploit_nNets,
308           inpParams.nnExploit_inputVectorSize,
309           inpParams.nnExploit_hiddenLayerSizes,
310           inpParams.nnExploit_outputVectorSize
311           );
312     #endif
313
314     pNNExploit->initializeRMSProp(inpParams.nnExploit_rmsProp_stepSize,
315                 inpParams.nnExploit_rmsProp_alpha,
316                 inpParams.nnExploit_rmsProp_eps,
317                 inpParams.nnExploit_rmsProp_maxEpochs*inpParams.buf_nTrainTestSamples,
318                 inpParams.nnExploit_rmsProp_tolerance,
319                 inpParams.nnExploit_rmsProp_shuffle
320                 );
321     nnExploit.push_back(pNNExploit);
322
323     #if NSE == 1
324     LearnNSEPredictor<TwoLayerNetwork,optimization::RMSprop> * pNNExploitTrainer =
325       new LearnNSEPredictor<TwoLayerNetwork,optimization::RMSprop>(//new NeuralNetworkPredictor<
      TwoLayerNetwork,optimization::RMSprop>(
326         inpParams.nnExploit_nNets,
327         inpParams.nnExploit_inputVectorSize,
328         inpParams.nnExploit_hiddenLayerSizes,
329         inpParams.nnExploit_outputVectorSize,
330         inpParams.sigmoidSlope,
331         inpParams.sigmoidThresh,
332         inpParams.errorThresh
333         );
334     #else
335       NeuralNetworkPredictor<TwoLayerNetwork,optimization::RMSprop> * pNNExploitTrainer =
336       new NeuralNetworkPredictor<TwoLayerNetwork,optimization::RMSprop>(
337           inpParams.nnExploit_nNets,
338           inpParams.nnExploit_inputVectorSize,
339           inpParams.nnExploit_hiddenLayerSizes,
340           inpParams.nnExploit_outputVectorSize
341           );
342     #endif
343
344
345     pNNExploitTrainer->initializeRMSProp(inpParams.nnExploit_rmsProp_stepSize,
346                 inpParams.nnExploit_rmsProp_alpha,
347                 inpParams.nnExploit_rmsProp_eps,
348                 inpParams.nnExploit_rmsProp_maxEpochs*inpParams.buf_nTrainTestSamples,
349                 inpParams.nnExploit_rmsProp_tolerance,
350                 inpParams.nnExploit_rmsProp_shuffle
351                 );
352     nnExploitTrainer.push_back(pNNExploitTrainer);
353   }
354
```

```
}

// Logic to select action in cognitive engine.
int RLNNCognitiveEngine::chooseAction() {
  double eProb;
  arma::mat predictions;
  std::vector<arma::mat> predictedActionVec;
  double perfThresh;
  int actionID;

  //set epsilon, nearbyhopidx
  if(_epsilon > _epsilonResetLim) {
    _epsilon = 1.0/((double) _epsilonIter);
    _epsilonIter++;
  } else { //reset epsilon
    _epsilon = 1.0;
    _epsilonIter=1;
  }

  //choose action
  if(_nnTrained && !_forceExplore) {
    eProb = math::Random();
    if(eProb <= _epsilon) { //exploration mode

      //generate the input vectors for prediction
      std::cout << "Exploring" << std::endl;
      #ifdef LOGGING
      logFile << "::Action Type: Exploring " << std::endl;
      #endif
      appSpecObj.genNNExploreInputs(_nnExploreInputs);

      //predict using input vectors. returns mean of all
      //parallel nets
      nnExplore.predict(_nnExploreInputs, predictions);

      //calculate current max threshold
      perfThresh = predictions.max() * (_nnExploreMaxPerfThresh);

      //if first explore after NN trained, then choose max perf.
      //randomly choose one of the actions (based on rejection rate)
      //and perfThresh
      if(_firstExploreAfterNNTrained) {
        std::cout << "Random Explore" << std::endl;
        actionID = predictions.index_max();
        _firstExploreAfterNNTrained = false;
      }
      else {
        arma::uvec idxsLessThresh = arma::find(predictions < perfThresh);
        arma::uvec idxsGreatThresh = arma::find(predictions >= perfThresh);
        //check edge case: if true, randomly choose (as they are all in the same
        //subset). else choose off of rejection rate.
        if((idxsLessThresh.n_elem==0) || (idxsGreatThresh.n_elem==0)) {
          std::cout << "Random Explore" << std::endl;
          actionID = _nActions;
          while(actionID >= _nActions) {
            actionID = (int) floor((double) _nActions*math::Random());
          }
        } else {
          if(math::Random() >= _nnRejectionRate) {
            //choose action worse than _nnRejectionRate
            int idx = idxsLessThresh.n_elem;
            while(idx >= idxsLessThresh.n_elem) {
              idx = (int) floor((double) idxsLessThresh.n_elem*math::Random());
            }
            actionID = idxsLessThresh(idx);
          } else { //choose action better than _nnRejectionRate
            int idx = idxsGreatThresh.n_elem;
            while(idx >= idxsGreatThresh.n_elem) {
              idx = (int) floor((double) idxsGreatThresh.n_elem*math::Random());
            }
            actionID = idxsGreatThresh(idx);
          }
        }
      }
      _exploitFlag = false;
    }
    else { //exploit mode
      //generate the input vectors for prediction
      std::cout << "Exploiting" << std::endl;
      #ifdef LOGGING
      logFile << "::Action Type: Exploiting " << std::endl;
      #endif
      appSpecObj.genNNExploitInputs(_nnExploitInputs);

      //predict using input vectors. returns mean of all
      //parallel nets
      for(int i=0; i<nnExploit.size(); i++) {
        nnExploit[i]->predict(_nnExploitInputs, predictions);
        predictedActionVec.push_back(predictions);
      }
      //the outputs of the nn's are processed using the application
      //specific object and the actionID corresponding to their outputs
```

```
447         //is returned back.
448         actionID = appSpecObj.processNNExploitOutputs(predictedActionVec);
449 
450         _exploitFlag = true;
451       }
452     }
453     else { //If reset or if first history buffer
454       //choose random action
455       if(_currentlyTraining) {
456         #ifdef LOGGING
457         logFile << "::Action Type: Exploiting " << std::endl;
458         #endif
459 
460         std::cout << "Exploiting" << std::endl;
461         //std::cout<<"training"<< std::endl;
462         actionID = appSpecObj.returnFallBackAction();
463         _exploitFlag = true;
464       } else {
465         #ifdef LOGGING
466         logFile << "::Action Type: Exploring " << std::endl;
467         #endif
468 
469         std::cout << "Random Explore" << std::endl;
470         actionID = _nActions;
471         while(actionID >= _nActions) {
472           actionID = (int) floor((float) _nActions*math::Random());
473         }
474         _exploitFlag = false;
475       }
476     }
477 
478     return actionID;
479 
480 }
481 // Logic for interpreting the evaluation metrics that are received.
482 // Also encompassses training when history buffer is full.
483 void RLNNCognitiveEngine::recordResponse(int actionID, const arma::rowvec &measurementVec) {
484     bool bufferFull;
485     arma::rowvec fitParams;
486     arma::colvec outVec;
487     double fitObserved;
488     static  boost::posix_time::ptime trainStartTime,trainEndTime;
489 
490     //process and save measurements from receiver
491     appSpecObj.processMeasurements(measurementVec);
492 
493     //compute new fitness observed
494     appSpecObj.getFitnessParams(fitParams);
495 
496     //This cout is here for demonstration purposes.
497     std::cout << "Normalized Objective Scores:" << std::endl;
498     std::cout << "Throughput: " << fitParams(0) << std::endl;
499     std::cout << "BER: " << fitParams(1) << std::endl;
500     std::cout << "Target BW: " << fitParams(2) << std::endl;
501     std::cout << "Spectral Efficiency: " << fitParams(3) << std::endl;
502     std::cout << "TX Power Efficiency: " << fitParams(4) << std::endl;
503     std::cout << "DC Power Consumed: " << fitParams(5) << std::endl;
504     fitObserved = arma::dot(_fitnessWeights,fitParams);
505     std::cout << "Multiobjective Fitness Score: " << fitObserved << std::endl;
506 
507     #ifdef LOGGING
508     logFile << "::Objective Fitnesses Observed: ";
509     logFile << fitParams;
510     logFile << "::Fitness Observed: " << fitObserved <<std::endl;
511     #endif
512 
513     appSpecObj.setFitnessObserved(fitObserved,_exploitFlag);
514 
515     //Adapt/updates NN_exploit input whenever forced exploration finds a better performance
516     if(_forceExplore) {
517       if(trBuf.getBufCntr()==0) {
518         _lastExploitFitObserved = fitObserved;
519       } else { //bufCtnr>0
520         if(fitObserved > _lastExploitFitObserved) {
521           _lastExploitFitObserved = fitObserved;
522         }
523       }
524     }
525 
526     //update nnExploit input whenever exploration finds a better performance
527     if(fitObserved >= _fitObservedMax) { //we found a new performance maximum
528       std::cout << "We found a new fObserved max." << std::endl;
529       _fitObservedMax = fitObserved;
530       if(!_exploitFlag) {
531         appSpecObj.updateNNExploitInputs();
532       }
533     }
534     else { //not a max observed fitness score
535       if(_exploitFlag) {
536         if(fitObserved<_lastExploitFitObserved) {
537           //Reset "More Efficient Mode". Threshold value is a hyperparameter (0.5 for specific
      missions, 0.1 for general)
```

```
538         if(   (((_lastExploitFitObserved-fitObserved)>_forceExploreThreshold) &&
539            (appSpecObj.lastAndNewNNExploitInputEqual()) &&
540            (appSpecObj.lastNNExploitInputEmpty()==false))
541            ||
542            (((_histRollBackCnt>=trBuf.getBufferSize()) &&
543            (appSpecObj.lastAndNewNNExploitInputEqual()) &&
544            (appSpecObj.lastNNExploitInputEmpty()==false)))
545           && (!_currentlyTraining)
546           ) {
547             _forceExplore = 1; //enter forced explore mode
548             _fitObservedMax = 0; //reset max tracking
549             trBuf.resetBuffer(); //reset NN history
550             _histRollBackCnt = 0;
551         }
552         // Quick "recover mode" using performances from the buffer.
553         //Triggers when 90% below previous exploration level
554         else if (fitObserved < _lastExploitFitObserved*0.9 ||  fitObserved < appSpecObj.
    getRollBackThreshold()) {
555           _histRollBackIdx++; //increment ptr
556           _histRollBackCnt++;
557           if(_histRollBackIdx==(trBuf.getBufCntr()-1)) { //wrap ptr around
558             _histRollBackIdx=0;
559           }
560
561           //get column in training buffer sorted by fitObserved, ascending order.
562           TrainingDataBuffer::InpOutBuffParams *par;
563           arma::colvec trainingInCol;
564           arma::colvec trainingOutCol;
565           par = appSpecObj.getNNInpOutBuffParms(1); //get exploit NN params
566           trBuf.buildTrainingColumn(1,_actionList,
567                       par->inpBuffIdxs,par->inpBuffExpandActions,par->inpNormParams,
568                       par->outBuffIdxs,par->outBuffExpandActions,par->outNormParams,
569                       appSpecObj.getFitObservedOutVecIdx(),(trBuf.getBufferSize()-1-
    _histRollBackIdx),
570                       trainingInCol, trainingOutCol);
571           //force update nnExploit Input
572           appSpecObj.forceSetNNExploitInputs(trainingInCol);
573         }
574         //accepts new exploitation peformance 90% above last exploittation threshold
575         else if(fitObserved > _lastExploitFitObserved*0.9 && _histRollBackIdx>-1) {
576           _lastExploitFitObserved = fitObserved;
577           appSpecObj.updateLastNNExploitInputs();
578             _histRollBackCnt = 0;
579         }
580         //if exploiting and current exploitation performance is worse than previous exploitation
     perf;
581           //roll back nn exploit input
582           else {
583             //std::cout <<"HERE4"<<std::endl;
584             appSpecObj.rollBackExploitInputs();
585               _histRollBackCnt = 0;
586           }
587       }
588       else { //update last exploitation performance
589         _lastExploitFitObserved = fitObserved;
590         appSpecObj.updateLastNNExploitInputs();
591         _histRollBackCnt = 0;
592       }
593     }
594   }
595
596   //push action/result into training data buffer
597   appSpecObj.genTrainingSample(outVec);
598   bufferFull = trBuf.addTrainingSample(actionID, outVec);
599
600   //train NNs if history buffer is full.
601   if(bufferFull) {
602     if(true) {
603       if(!_currentlyTraining) {
604         std::cout << "Training Occurring" << std::endl;
605         trainStartTime = boost::posix_time::microsec_clock::local_time();
606         _currentlyTraining = true;
607         _trainingComplete = false;
608
609         //build the nnExplore(0) and nnExploit(1) training sets
610         std::vector<arma::mat> * nnInputs;
611         std::vector<arma::mat> * nnOutputs;
612         std::cout << "Buffer Full Thread" << std::endl;
613         for(int i=0; i<2; i++) {
614           TrainingDataBuffer::InpOutBuffParams *par;
615           par = appSpecObj.getNNInpOutBuffParms(i);
616           trBuf.buildTrainingSet(i,_actionList,
617                       par->inpBuffIdxs,par->inpBuffExpandActions,par->inpNormParams,
618                       par->outBuffIdxs,par->outBuffExpandActions,par->outNormParams
619                       );
620         }
621         //retrieve the buffers
622         nnInputs = trBuf.getTrainTestInput();
623         nnOutputs = trBuf.getTrainTestOutput();
624
625
626         tTrainingVec.push_back(std::thread ([this,nnInputs,nnOutputs](){
```

```
627
628             //TRAIN NN_EXPLORE
629             nnExploreTrainer.train((*nnInputs)[0],(*nnOutputs)[0],_trainFrac);
630
631             //train NN array
632             for(int i=0; i<nnExploit.size(); i++) {
633               nnExploitTrainer[i]->train((*nnInputs)[1],(*nnOutputs)[1].row(i),_trainFrac);
634             }
635             _trainingComplete = true;
636           }));
637
638         }
639         if(_trainingComplete) {
640           //join thread
641           tTrainingVec[0].join();
642           tTrainingVec.pop_back();
643           _currentlyTraining = false; /**********************/
644
645           //transfer weights
646           arma::mat trainedWeights;
647           nnExploreTrainer.exportWeights(trainedWeights);
648           nnExplore.importWeights(trainedWeights);
649           for(int i=0; i<nnExploit.size(); i++) {
650             nnExploitTrainer[i]->exportWeights(trainedWeights);
651             nnExploit[i]->importWeights(trainedWeights);
652           }
653
654           //flag as initially trained
655           _nnTrained = true;
656           _forceExplore = false; //allow us to use NNs
657           _histRollBackIdx = 0;
658           _firstExploreAfterNNTrained=true;
659           //prune buffer
660           trBuf.pruneDataBuffer(_pruneFrac);
661
662         } else {
663           std::cout << "Training Occurring" << std::endl;
664
665           #ifdef LOGGING
666           logFile << "::Training: Yes" <<std::endl;
667           #endif
668         }
669       }
670     } else {
671       #ifdef LOGGING
672       logFile << "::Training: No" <<std::endl;
673       #endif
674     }
675 }
```

D.3 NeuralNetworkPredictor.cpp

```
#ifndef NEURALNETWORKPREDICTOR
#define NEURALNETWORKPREDICTOR

#include <mlpack/core.hpp>

#include <mlpack/methods/ann/activation_functions/logistic_function.hpp>
#include <mlpack/methods/ann/activation_functions/identity_function.hpp>

#include <mlpack/methods/ann/init_rules/random_init.hpp>

#include <mlpack/methods/ann/layer/linear_layer.hpp>
#include <mlpack/methods/ann/layer/base_layer.hpp>
#include <mlpack/methods/ann/layer/identity_output_layer.hpp>

#include <mlpack/methods/ann/ffn.hpp>
#include <mlpack/methods/ann/performance_functions/mse_function.hpp>
#include <mlpack/core/optimizers/rmsprop/rmsprop.hpp>
#include <mlpack/core/optimizers/gradient_descent/gradient_descent.hpp>
#include <mlpack/core/optimizers/adadelta/ada_delta.hpp>
#include <mlpack/core/optimizers/sgd/sgd.hpp>

#include "/home/tim/Desktop/rlnn5/ThreeLayerNetwork.cpp"
#include "/home/tim/Desktop/rlnn5/TwoLayerNetwork.cpp"
#include "/home/tim/Desktop/rlnn5/Logging.hpp"
#include "/home/tim/Desktop/rlnn5/FeedForwardNetwork.cpp"
#include "/home/tim/Desktop/rlnn5/MemoryManagement.hpp"
#include "/home/tim/Desktop/rlnn5/RecursiveLMHelper.cpp"

#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <mlpack/prereqs.hpp>
#include <fstream>
#include <boost/archive/tmpdir.hpp>
#include <boost/serialization/base_object.hpp>
#include <boost/serialization/utility.hpp>
#include <boost/serialization/list.hpp>
#include <boost/serialization/assume_abstract.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/serialization/serialization.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>

#include <iostream>

using namespace mlpack;

namespace boost {
namespace serialization {
class access;
}
}

template <class NetType, template <class T> class OptType>
class NeuralNetworkPredictor {
  public:
    std::vector<NetType *> nets;

    void initializeRMSProp(double stepSize, double alpha, double eps, size_t maxIterations,
     double tolerance, bool shuffle);
    void train(const arma::mat &trainData, const arma::mat &trainLabels, double trainDataFrac);
    void predict(arma::mat &inputData, arma::mat &prediction);
    void exportWeights(arma::mat &weights);
    void importWeights(arma::mat &weights);
    void loadOldRun(std::string filename);
    void saveCurrentRun(std::string filename);

    NeuralNetworkPredictor(int nNets, int inputVectorSize, const std::vector<int>
     hiddenLayerSize, int outputVectorSize);

  private:
    std::vector<OptType<decltype(((NetType*)nullptr)->net)> *> _opts;
    arma::rowvec _mse_perfs;
    arma::mat _weights;
    arma::colvec _initWeights;

    // Serialization used for saving runs after completed.
    friend class boost::serialization::access;

    template<class Archive>
    void serialize(Archive & ar, const int version) {
      ar & _mse_perfs;
      ar & _weights;
      ar & _initWeights;
    }

    std::vector<FeedForwardNetwork *> nnFFNVec;
};
// Initialization of NeuralNetPredictor type.
template <class NetType, template <class T> class OptType>
```

```
87 NeuralNetworkPredictor<NetType,OptType>::NeuralNetworkPredictor(int nNets, int inputVectorSize,
       const std::vector<int> hiddenLayerSize, int outputVectorSize) {
88   for(int i=0; i<nNets; i++) {
89     NetType * t = new NetType(inputVectorSize,hiddenLayerSize,outputVectorSize);
90     OptType<decltype(t->net)> * op = new OptType<decltype(t->net)>(t->net);
91     nets.push_back(t);
92     _opts.push_back(op);
93
94     FeedForwardNetwork * pFFN = new FeedForwardNetwork;
95     nnFFNVec.push_back(pFFN);
96   }
97
98   _mse_perfs.set_size(nNets);
99
100  //initialize NNs for LM training
101  std::vector<int> networkSize;
102  networkSize.resize(hiddenLayerSize.size());
103  for(int i=0; i<hiddenLayerSize.size(); i++) {
104    networkSize[i] = hiddenLayerSize[i];
105  }
106  networkSize.push_back(outputVectorSize);
107
108  //init first network to grab initial weights
109  if(hiddenLayerSize.size()==2) {
110    const std::vector<std::string> actTypePerLayer = {"logsig","logsig","linear"};
111    nnFFNVec[0]->initNetwork(inputVectorSize,networkSize.size(),networkSize,actTypePerLayer);
112  }
113  else {
114    const std::vector<std::string> actTypePerLayer = {"logsig","linear"};
115    nnFFNVec[0]->initNetwork(inputVectorSize,networkSize.size(),networkSize,actTypePerLayer);
116  }
117  nnFFNVec[0]->exportWeights(_initWeights);
118
119  //build rest of parallel networks.  All parallel NNs have the same init weights.  the "
       randomness"
120  //comes from uniquely shuffled train/validation data.
121  if(hiddenLayerSize.size()==2) {
122    const std::vector<std::string> actTypePerLayer = {"logsig","logsig","linear"};
123    for(int i=1; i<nnFFNVec.size(); i++) {
124      nnFFNVec[i]->initNetwork(inputVectorSize,networkSize.size(),networkSize,actTypePerLayer);
125      nnFFNVec[i]->importWeights(_initWeights);
126    }
127  }
128  else {
129    const std::vector<std::string> actTypePerLayer = {"logsig","linear"};
130    for(int i=1; i<nnFFNVec.size(); i++) {
131      nnFFNVec[i]->initNetwork(inputVectorSize,networkSize.size(),networkSize,actTypePerLayer);
132      nnFFNVec[i]->importWeights(_initWeights);
133    }
134  }
135 }
136
137 // Initialization of RMSProp necessary for using MLPacks network. Never actually used.
138 template <class NetType, template <class T> class OptType>
139 void NeuralNetworkPredictor<NetType,OptType>::initializeRMSProp(double stepSize, double alpha,
       double eps, size_t maxIterations, double tolerance, bool shuffle) {
140   for(int i=0; i<_opts.size(); i++) {
141     _opts[i]->Alpha() = alpha;
142     _opts[i]->Epsilon() = eps;
143     _opts[i]->MaxIterations() = maxIterations;
144     _opts[i]->Shuffle() = shuffle;
145     _opts[i]->StepSize() = stepSize;
146     _opts[i]->Tolerance() = tolerance;
147   }
148 }
149
150 // Train ensemble.
151 template <class NetType, template <class T> class OptType>
152 void NeuralNetworkPredictor<NetType,OptType>::train(const arma::mat &trainData, const arma::mat
       &trainLabels, double trainDataFrac) {
153   arma::mat prediction;
154   arma::mat shuffledTrainData;
155   arma::mat shuffledTrainLabels;
156   arma::mat shuffledValData;
157   arma::mat shuffledValLabels;
158
159   //Shuffle data
160   shuffledTrainData.set_size(trainData.n_rows,
161     (int) floor(trainData.n_cols * trainDataFrac));
162   shuffledTrainLabels.set_size(trainLabels.n_rows,
163     (int) floor(trainLabels.n_cols * trainDataFrac));
164   shuffledValData.set_size(trainData.n_rows,
165     (int) (trainData.n_cols - floor(trainData.n_cols * trainDataFrac)));
166   shuffledValLabels.set_size(trainLabels.n_rows,
167     (int) (trainLabels.n_cols - floor(trainLabels.n_cols * trainDataFrac)));
168
169   //train NNs
170   arma::mat weightsMat;
171   for(int i=0; i<nets.size(); i++) {
172
173     arma::colvec shuffledOrder = arma::regspace(0,1,trainData.n_cols -1);
174     shuffledOrder = arma::shuffle(shuffledOrder);
```

```
175     for(int i=0; i<trainData.n_cols; i++) {
176       if(i < (int) floor(trainData.n_cols * trainDataFrac)) {
177         shuffledTrainData.col(i) = trainData.col(shuffledOrder(i));
178         shuffledTrainLabels.col(i) = trainLabels.col(shuffledOrder(i));
179       } else {
180         shuffledValData.col(i-((int) floor(trainData.n_cols * trainDataFrac))) =
181           trainData.col(shuffledOrder(i));
182         shuffledValLabels.col(i-((int) floor(trainLabels.n_cols * trainDataFrac))) =
183           trainLabels.col(shuffledOrder(i));
184       }
185     }
186
187     //init weights to same init values each time and train using LM
188     nnFFNVec[i]->importWeights(_initWeights);
189     //Run LM or RLM depending on configuration.
190     #if LM==1
191       nnFFNVec[i]->runLM(shuffledTrainData,shuffledTrainLabels,shuffledValData,shuffledValLabels
       ,0.0,1e-12,1e10,500,20);
192
193     #elif RLM==1
194       nnFFNVec[i]->runRLM(shuffledTrainData,shuffledTrainLabels,shuffledValData,
       shuffledValLabels,0.0,1e-12,1e10,500,20);
195     #endif
196
197     //update weights in MLPack NN
198     arma::colvec weightsCol;
199     nnFFNVec[i]->exportWeights(weightsCol);
200     if(i==0) {
201       weightsMat.set_size(weightsCol.n_elem,nets.size());
202     }
203     weightsMat.col(i) = weightsCol;
204   }
205
206   importWeights(weightsMat);
207 }
208
209 // Predict with network.
210 template <class NetType, template <class T> class OptType>
211 void NeuralNetworkPredictor<NetType,OptType>::predict(arma::mat &inputData, arma::mat &
       prediction) {
212   arma::mat tmpPrediction;
213
214   for(int i=0; i<nets.size(); i++) {
215     nets[i]->net.Predict(inputData,tmpPrediction);
216     if(i==0) {
217       prediction = tmpPrediction;
218     } else {
219       prediction = tmpPrediction + prediction;
220     }
221   }
222   prediction = prediction/nets.size();
223   prediction = arma::clamp(prediction,0,1);
224
225 }
226
227 /*********Functions that enable serialization**********/
228 template <class NetType, template <class T> class OptType>
229 void NeuralNetworkPredictor<NetType,OptType>::exportWeights(arma::mat &weights) {
230   arma::mat tmpWeights = nets[0]->net.Parameters();
231   weights.set_size(tmpWeights.n_elem,nets.size());
232   for(int i=0; i<nets.size(); i++) {
233     weights.col(i) = nets[i]->net.Parameters();
234   }
235
236 }
237
238 template <class NetType, template <class T> class OptType>
239 void NeuralNetworkPredictor<NetType,OptType>::importWeights(arma::mat &weights) {
240
241   for(int i=0; i<nets.size(); i++) {
242     arma::mat &currentWeights = nets[i]->net.Parameters();
243     currentWeights = weights.col(i);
244   }
245
246 }
247
248 template <class NetType, template <class T> class OptType>
249 void NeuralNetworkPredictor<NetType,OptType>::saveCurrentRun(std::string filename) {
250   exportWeights(_weights);
251   std::ofstream ofs(filename);
252   //save data to archvie
253   boost::archive::text_oarchive oa(ofs);
254   //write class instance to archive
255   oa & *this;
256   //archive and stream closed with destructors are called
257 };
258
259 template <class NetType, template <class T> class OptType>
260 void NeuralNetworkPredictor<NetType,OptType>::loadOldRun(std::string filename) {
261
262   std::ifstream ifs(filename);
263   boost::archive::text_iarchive ia(ifs);
```

```
264     //read class state from archive
265     ia & *this;
266     //archive and stream closed with destructors are called.
267
268     importWeights(_weights);
269 }
270 #endif
```

D.4 LearnNSEPredictor.cpp

```
1
2  #include <mlpack/core.hpp>
3
4  #include <mlpack/methods/ann/activation_functions/logistic_function.hpp>
5  #include <mlpack/methods/ann/activation_functions/identity_function.hpp>
6
7  #include <mlpack/methods/ann/init_rules/random_init.hpp>
8
9  #include <mlpack/methods/ann/layer/linear_layer.hpp>
10 #include <mlpack/methods/ann/layer/base_layer.hpp>
11 #include <mlpack/methods/ann/layer/identity_output_layer.hpp>
12
13 #include <mlpack/methods/ann/ffn.hpp>
14 #include <mlpack/methods/ann/performance_functions/mse_function.hpp>
15 #include <mlpack/core/optimizers/rmsprop/rmsprop.hpp>
16 #include <mlpack/core/optimizers/gradient_descent/gradient_descent.hpp>
17 #include <mlpack/core/optimizers/adadelta/ada_delta.hpp>
18 #include <mlpack/core/optimizers/sgd/sgd.hpp>
19
20 #include "/home/tim/Desktop/rlnn5/ThreeLayerNetwork.cpp"
21 #include "/home/tim/Desktop/rlnn5/TwoLayerNetwork.cpp"
22 #include "/home/tim/Desktop/rlnn5/Logging.cpp"
23 #include "/home/tim/Desktop/rlnn5/FeedForwardNetwork.cpp"
24
25 #include <boost/archive/text_oarchive.hpp>
26 #include <boost/archive/text_iarchive.hpp>
27 #include <mlpack/prereqs.hpp>
28 #include <fstream>
29 #include <boost/archive/tmpdir.hpp>
30 #include <boost/serialization/base_object.hpp>
31 #include <boost/serialization/utility.hpp>
32 #include <boost/serialization/list.hpp>
33 #include <boost/serialization/assume_abstract.hpp>
34 #include <boost/serialization/vector.hpp>
35 #include <boost/serialization/serialization.hpp>
36
37 #include <iostream>
38 using namespace mlpack;
39
40 namespace boost {
41 namespace serialization {
42 class access;
43 }
44 }
45
46 /******Helper functions, external of LearnNSEPredictor class*****/
47 // Circular shift used in circular buffer calculations.
48 void circularShift(arma::rowvec vecToShift,arma::rowvec& retVec, int shiftNum){
49
50   // #ifdef LOGGING
51   //  logFile << "Circular shift" <<std::endl;
52   // #endif
53   retVec.set_size(vecToShift.n_elem);
54
55   for (int i = 0; i < vecToShift.n_elem; i++){
56     retVec[i] = vecToShift((i+shiftNum)%vecToShift.n_elem);
57   }
58 }
59
60 // MSE calculation.
61 arma::rowvec squaredError(arma::rowvec vector1, arma::rowvec vector2){
62   arma::rowvec err = vector1 - vector2;
63   return err % err;
64 }
65
66 template <class NetType, template <class T> class OptType>
67 class LearnNSEPredictor {
68   public:
69     std::vector<NetType *> nets;
70
71     void initializeRMSProp(double stepSize, double alpha, double eps, size_t maxIterations,
       double tolerance, bool shuffle);
72     void train(const arma::mat &trainData, const arma::mat &trainLabels, double trainDataFrac);
73     void predict(arma::mat &inputData, arma::mat &prediction);
74     void exportWeights(arma::mat &weights);
75     void importWeights(arma::mat &weights);
76     void loadOldRun(std::string filename);
77     void saveCurrentRun(std::string filename);
78
79     LearnNSEPredictor(int nNets,
80               int inputVectorSize,
81               const std::vector<int> hiddenLayerSize,
82               int outputVectorSize,
83               double sigmoidSlope,
84               double sigmoidThresh,
85               double errThresh
86               );
87
```

```
88    private:
89      std::vector<OptType<decltype(((NetType*)nullptr)->net)> *> _opts;
90      arma::rowvec _mse_perfs;
91      arma::mat _weights;
92      arma::colvec _initWeights;
93      arma::colvec netWeights;
94      arma::mat beta;
95      int betaInd;
96      double sigmoidSlope,sigmoidThresh,errThresh;
97      bool initialized;
98      long trainingCounter;
99      friend class boost::serialization::access;
100
101     template<class Archive>
102     void serialize(Archive & ar, const int version) {
103       ar & _mse_perfs;
104       ar & _weights;
105       ar & _initWeights;
106     }
107
108     std::vector<FeedForwardNetwork *> nnFFNVec;
109 };
110
111 // Class Initialization
112 template <class NetType, template <class T> class OptType>
113 LearnNSEPredictor<NetType,OptType>::LearnNSEPredictor(int nNets,
114                             int inputVectorSize,
115                             const std::vector<int> hiddenLayerSize,
116                             int outputVectorSize,
117                             double sigmoidSlopeTmp,
118                             double sigmoidThreshTmp,
119                             double errThreshTmp)
120 {
121   //Set LearnNSE Parameters.
122   initialized = false;
123   trainingCounter = 0;
124   betaInd = 0;
125   sigmoidThresh = sigmoidThreshTmp;
126   sigmoidSlope = sigmoidSlopeTmp;
127   errThresh = errThreshTmp;
128   beta = arma::ones(nNets,nNets);
129   netWeights = arma::ones(nNets);
130
131   // Intiialize list of networks.
132   for(int i=0; i<nNets; i++) {
133     NetType * t = new NetType(inputVectorSize,hiddenLayerSize,outputVectorSize);
134     OptType<decltype(t->net)> * op = new OptType<decltype(t->net)>(t->net);
135     nets.push_back(t);
136     _opts.push_back(op);
137
138     FeedForwardNetwork * pFFN = new FeedForwardNetwork;
139     nnFFNVec.push_back(pFFN);
140   }
141
142   _mse_perfs.set_size(nNets);
143
144   //initialize my NNs for LM training
145   std::vector<int> networkSize;
146   networkSize.resize(hiddenLayerSize.size());
147   for(int i=0; i<hiddenLayerSize.size(); i++) {
148     networkSize[i] = hiddenLayerSize[i];
149   }
150   networkSize.push_back(outputVectorSize);
151
152   //init first network to grab initial weights
153   if(hiddenLayerSize.size()==2) {
154     const std::vector<std::string> actTypePerLayer = {"logsig","logsig","linear"};
155     nnFFNVec[0]->initNetwork(inputVectorSize,networkSize.size(),networkSize,actTypePerLayer);
156   }
157   else {
158     const std::vector<std::string> actTypePerLayer = {"logsig","linear"};
159     nnFFNVec[0]->initNetwork(inputVectorSize,networkSize.size(),networkSize,actTypePerLayer);
160   }
161   nnFFNVec[0]->exportWeights(_initWeights);
162
163   //build rest of parallel networks.  All parallel NNs have the same init weights.  the "
        randomness"
164   //comes from uniquely shuffled train/validation data.
165   if(hiddenLayerSize.size()==2) {
166     const std::vector<std::string> actTypePerLayer = {"logsig","logsig","linear"};
167     for(int i=1; i<nnFFNVec.size(); i++) {
168       nnFFNVec[i]->initNetwork(inputVectorSize,networkSize.size(),networkSize,actTypePerLayer);
169       nnFFNVec[i]->importWeights(_initWeights);
170     }
171   }
172   else {
173     const std::vector<std::string> actTypePerLayer = {"logsig","linear"};
174     for(int i=1; i<nnFFNVec.size(); i++) {
175       nnFFNVec[i]->initNetwork(inputVectorSize,networkSize.size(),networkSize,actTypePerLayer);
176       nnFFNVec[i]->importWeights(_initWeights);
177     }
178   }
```

```
}
// Initialization of RMSProp necessary for using MLPacks network. Never actually used.
template <class NetType, template <class T> class OptType>
void LearnNSEPredictor<NetType,OptType>::initializeRMSProp(double stepSize, double alpha, double
     eps, size_t maxIterations, double tolerance, bool shuffle) {
  for(int i=0; i<_opts.size(); i++) {
    _opts[i]->Alpha() = alpha;
    _opts[i]->Epsilon() = eps;
    _opts[i]->MaxIterations() = maxIterations;
    _opts[i]->Shuffle() = shuffle;
    _opts[i]->StepSize() = stepSize;
    _opts[i]->Tolerance() = tolerance;
  }
}

// Train Ensemble
template <class NetType, template <class T> class OptType>
void LearnNSEPredictor<NetType,OptType>::train(const arma::mat &trainData, const arma::mat &
     trainLabels, double trainDataFrac){
  arma::mat prediction,prediction_eval,prediction_tmp;
  arma::mat shuffledTrainData;
  arma::mat shuffledTrainLabels;
  arma::mat shuffledValData;
  arma::mat shuffledValLabels;
  arma::colvec Bt_sampBySamp,Dt_sampBySamp, Wt_sampBySamp;
  arma::colvec weightsCol;
  arma::colvec omega;
  arma::rowvec omega_trans;
  arma::rowvec betaVec;
  arma::rowvec betaTmp;
  arma::colvec netWeightsTmp;

  netWeightsTmp.set_size(arma::size(netWeights));
  arma::colvec sigSlopeVec,b,mseInit;
  double Bt, Et;
  double epsilon_tk,beta_hat;
  int mt,netInd,indMax,i;
  int nNets = nets.size();
  bool firstTrainingBatch;

  if(trainingCounter < nNets)
    firstTrainingBatch = true;
  else
    firstTrainingBatch = false;

  //resize shuffle buffers
  shuffledTrainData.set_size(trainData.n_rows,
    (int) floor(trainData.n_cols * trainDataFrac));
  shuffledTrainLabels.set_size(trainLabels.n_rows,
    (int) floor(trainLabels.n_cols * trainDataFrac));
  shuffledValData.set_size(trainData.n_rows,
    (int) (trainData.n_cols - floor(trainData.n_cols * trainDataFrac)));
  shuffledValLabels.set_size(trainLabels.n_rows,
    (int) (trainLabels.n_cols - floor(trainLabels.n_cols * trainDataFrac)));

  //shuffle data and split into train/val sets
  arma::colvec shuffledOrder = arma::regspace(0,1,trainData.n_cols -1);
  shuffledOrder = arma::shuffle(shuffledOrder);
  for(i=0; i<trainData.n_cols; i++) {
    if(i < (int) floor(trainData.n_cols * trainDataFrac)) {
      shuffledTrainData.col(i) = trainData.col(shuffledOrder(i));
      shuffledTrainLabels.col(i) = trainLabels.col(shuffledOrder(i));
    } else {
      shuffledValData.col(i-((int) floor(trainData.n_cols * trainDataFrac))) =
        trainData.col(shuffledOrder(i));
      shuffledValLabels.col(i-((int) floor(trainLabels.n_cols * trainDataFrac))) =
        trainLabels.col(shuffledOrder(i));
    }
  }


  //if net.initialized == false, net.beta = []; end
  mt = shuffledTrainData.n_cols;
  Dt_sampBySamp = arma::ones<arma::colvec>(mt)/mt;

  //If this isn't the first network.
  if(initialized){
    // Step 1: Compute error of existing ensemble on new data.
    predict(shuffledTrainData,prediction);
    //compute mean squared error
    mseInit = squaredError(prediction,shuffledTrainLabels) / shuffledTrainData.n_cols;

    Et = arma::accu(mseInit);
    // Get beta for each network.
    Bt = Et / (1-Et);
    Bt_sampBySamp = mseInit/(1-mseInit);
    if(Bt == 0) Bt = 1/mt;

    //Update and normalize instance weights.
    Wt_sampBySamp = 1/mt * Bt_sampBySamp;
  }

```

```
269   // Step 2: Create new Predictor
270   // If the ensemble isn't full, just use first unused FFNN. Otherwise, cycle through
        overwriting the oldest.
271   if (firstTrainingBatch)
272     netInd = trainingCounter;
273   else
274     netInd = trainingCounter % nNets;
275
276   arma::mat weightsMat;
277   exportWeights(weightsMat);
278   nnFFNVec[netInd]->importWeights(_initWeights);
279   nnFFNVec[netInd]->runLM(shuffledTrainData,shuffledTrainLabels,shuffledValData,
        shuffledValLabels,0.0,1e-12,1e10,500,20);
280
281   //update weights in MLPack NN
282   nnFFNVec[netInd]->exportWeights(weightsCol);
283   weightsMat.col(netInd) = weightsCol;
284
285
286   importWeights(weightsMat);
287   predict(shuffledTrainData,prediction_eval);
288
289   if(trainingCounter < nNets)
290     indMax = trainingCounter + 1;
291   else
292     indMax = nNets;
293   float tmpFloat;
294
295   for( i = 0; i < indMax; i++){
296     epsilon_tk = arma::accu(Dt_sampBySamp % squaredError(prediction_eval,shuffledTrainLabels).t
        ())/mt;
297     // #ifdef LOGGING
298     //  logFile << "F5" <<std::endl;
299     // #endif
300     tmpFloat = arma::accu(squaredError(prediction_eval,shuffledTrainLabels))/mt;
301     // #ifdef LOGGING
302     //  logFile << "F6" <<std::endl;
303     // #endif
304     if(epsilon_tk > 0.5){
305
306       if((i<netInd && firstTrainingBatch) || (i != netInd && !firstTrainingBatch))
307         epsilon_tk = 0.5; // if old network keep
308       else if(i == netInd) { //Retrain
309         nnFFNVec[netInd]->importWeights(_initWeights);
310         nnFFNVec[netInd]->runLM(shuffledTrainData,shuffledTrainLabels,shuffledValData,
        shuffledValLabels,0.0,1e-12,1e10,500,20);
311
312         //arma::colvec weightsCol;
313         // nnFFNVec[netInd]->exportWeights(weightsCol);
314         /*******************/
315         // if(i==0) {
316         //  weightsMat.set_size(weightsCol.n_elem,nets.size());
317         // }
318         // weightsMat.col(i) = weightsCol;
319
320         // importWeights(weightsMat);
321       }
322     }
323
324     // #ifdef LOGGING
325     //  logFile << "F7,trainingCounter: "<<trainingCounter <<std::endl;
326     // #endif
327     beta(betaInd,i) = epsilon_tk /(1-epsilon_tk);
328   }
329   // Step 4: Compute classifier Weights.
330   if (trainingCounter == 0){
331     if (beta(betaInd,trainingCounter) < errThresh)
332       beta(betaInd,trainingCounter) = errThresh;
333
334     netWeights[0] = log(1/beta(betaInd,trainingCounter));
335   }
336   else{
337
338     // Vectorization of single value to make eltwise mult work.
339     sigSlopeVec = sigmoidSlope * arma::ones(omega.n_elem);
340
341     for(i = 0; i < indMax; i++){
342       if (trainingCounter < nNets)
343         omega = arma::regspace(1,trainingCounter-i + 1);
344       else
345         omega = arma::regspace(1,nNets);
346
347       // Recalculate b
348       b = (nNets - i - sigmoidThresh) * arma::ones(omega.n_elem);
349
350       // Update omega
351       omega = 1/(1+exp(sigSlopeVec % (omega - b)));
352       omega = omega/arma::accu(omega);
353       omega_trans = omega.t();
354       betaVec = beta.row(i);
355
356       // update beta prediction (used in determining network weighting.
```

```
357       if(firstTrainingBatch){
358         beta_hat = arma::sum(omega_trans % (betaVec.subvec(i,trainingCounter)));
359       }
360       else {
361         // Circular shift so that the beta lines up correctly.
362         circularShift(betaVec,betaTmp,nNets-betaInd);
363         beta_hat = arma::accu(omega_trans % betaTmp);
364       }
365       if (beta_hat < errThresh)
366         beta_hat = errThresh;
367       netWeights[i] = log(1/beta_hat);
368     }
369   }
370   // Normalize network weights to sum to 1.
371   for(int i = 0; i < indMax; i++){
372     netWeightsTmp[i] = netWeights[i] / arma::accu(netWeights.subvec(0,indMax-1));
373   }
374   // Copy netweights back.
375   for(int i = 0; i < indMax; i++){
376     netWeights[i] = netWeightsTmp[i];
377   }
378
379   // Increment update for next iteration.
380   trainingCounter += 1;
381   betaInd = (betaInd + 1) % nNets;
382
383 }
384
385 template <class NetType, template <class T> class OptType>
386 void LearnNSEPredictor<NetType,OptType>::predict(arma::mat &inputData, arma::mat &prediction) {
387   arma::mat tmpPrediction;
388   int nNets = nets.size();
389   int indMax;
390   if (trainingCounter < nNets)
391     indMax = trainingCounter + 1;
392   else
393     indMax = nNets;
394
395   for(int i=0; i<indMax; i++) {
396     nets[i]->net.Predict(inputData,tmpPrediction);
397
398     if(i==0) {
399       prediction = tmpPrediction * netWeights[i];
400     } else {
401       prediction = tmpPrediction * netWeights[i] + prediction;
402     }
403   }
404   prediction = arma::clamp(prediction,0,1);
405
406 }
407
408 /******Functions enabling serialization*****/
409 template <class NetType, template <class T> class OptType>
410 void LearnNSEPredictor<NetType,OptType>::exportWeights(arma::mat &weights) {
411   arma::mat tmpWeights = nets[0]->net.Parameters();
412   weights.set_size(tmpWeights.n_elem,nets.size());
413   for(int i=0; i<nets.size(); i++) {
414     weights.col(i) = nets[i]->net.Parameters();
415   }
416
417 }
418
419 template <class NetType, template <class T> class OptType>
420 void LearnNSEPredictor<NetType,OptType>::importWeights(arma::mat &weights) {
421
422   for(int i=0; i<nets.size(); i++) {
423     arma::mat &currentWeights = nets[i]->net.Parameters();
424     currentWeights = weights.col(i);
425   }
426
427 }
428
429 template <class NetType, template <class T> class OptType>
430 void LearnNSEPredictor<NetType,OptType>::saveCurrentRun(std::string filename) {
431   exportWeights(_weights);
432   std::ofstream ofs(filename);
433   //save data to archvie
434   boost::archive::text_oarchive oa(ofs);
435   //write class instance to archive
436   oa & *this;
437   //archive and stream closed with destructors are called
438 };
439
440 template <class NetType, template <class T> class OptType>
441 void LearnNSEPredictor<NetType,OptType>::loadOldRun(std::string filename) {
442
443   std::ifstream ifs(filename);
444   boost::archive::text_iarchive ia(ifs);
445   //read class state from archive
446   ia & *this;
447   //archive and stream closed with destructors are called.
448
```

```
449     importWeights(_weights);
450 }
```

D.5 FeedForwardNeuralNetwork.cpp

```
#ifndef FEEDFORWARDNETWORK
#define FEEDFORWARDNETWORK

#include <vector>
#include <cmath>
#include <string>
#include <iostream>
#include <mlpack/core.hpp>
#include "/home/tim/Desktop/rlnn5/Logging.cpp"
#include "/home/tim/Desktop/rlnn5/RecursiveLMHelper.cpp"
#include <boost/date_time/posix_time/posix_time.hpp>

class FeedForwardNetwork {
public:

  void runLM( const arma::mat &inpPatternTrain,
        const arma::mat &outPatternTrain,
        const arma::mat &inpPatternValid,
        const arma::mat &outPatternValid,
        double minError,
        double minGrad,
        double maxMu,
        int maxIterations,
        int maxValidationFails
      );

  void runRLM(const arma::mat &inpPatternTrain,
        const arma::mat &outPatternTrain,
        const arma::mat &inpPatternValid,
        const arma::mat &outPatternValid,
        double minError,
        double minGrad,
        double maxMu,
        int maxIterations,
        int maxValidationFails
      );
  void forwardPropagate(const arma::colvec &inputs,
              arma::colvec &outputs
             );

  void initNetwork(int nInputs, int nLayers,
          std::vector<int> neuronsPerLayer,
          std::vector<std::string> activationTypePerLayer
          );
  void reInitNetwork();
  void exportWeights(arma::colvec &weights);
  void importWeights(const arma::colvec &weights);

  void printWeights(const arma::colvec &weights);
  void printJacobianMatrix(const arma::mat &jMat);
  void printErrorVec(const arma::colvec &errorVec);
  void printDeltaVals();
  void printSlopes();
  void printNeuronInputs();
  void printWeights();
  void printOutputs(const arma::colvec &outputs);

private:
  std::vector< std::vector<Neuron> > network_;
  arma::colvec trainedWeights_;
  RecursiveLMHelper rlmHelper_;

  void initNeuronWeights(std::vector<double> &weights);
  void applyInputs(const arma::colvec &inputs, std::vector<Neuron> &firstLayerNeurons);
  void applyLayer(std::vector<Neuron> &currentLayerNeurons, arma::colvec &outputs);
  void applyLayer(std::vector<Neuron> &currentLayerNeurons, std::vector<Neuron> &
    nextLayerNeurons);
  void calculateActivationFnSlope(const double &input, double &output, std::string type);
  void applyActivation(const double &input, double &output, std::string type);
  void applyWeights(const std::vector<double> &weights, const std::vector<double> &inputs,
    double &output);

  void updateNetworkWithWeights(std::vector<std::vector<Neuron>> &network,
              const arma::colvec &weights
             );
  double computeMeanSquareError(const arma::colvec &errorVec, int nOutputs, int nPatterns);
  void calculateRecursiveWeights(const std::vector<std::vector<Neuron>> &network,
              const arma::colvec &allCurrentWeights,
              const arma::colvec &errorVec,
              const arma::colvec &grad,
              double mu,
              arma::colvec &allNewWeights
             );
  void calculateWeightUpdate(const std::vector<std::vector<Neuron>> &network,
                const arma::mat &jMat,
                const arma::colvec &allCurrentWeights,
                const arma::colvec &errorVec,
                double mu,
```

```
87                  arma::colvec &allNewWeights
88                 );
89   void calculateJacobianMatrix(std::vector<std::vector<Neuron>> &network,
90                arma::mat &jMat,
91                arma::colvec &errorVec,
92                arma::colvec &allCurrentWeights,
93                const arma::mat &inpPattern,
94                const arma::mat &outPattern
95                );
96   void backwardPropagate(std::vector<std::vector<Neuron>> &network,
97                const arma::colvec &actualoutputVec,
98                const arma::colvec &correctOutputVec,
99                arma::colvec &errorVec
100               );
101  void forwardPropagate(std::vector<std::vector<Neuron>> &network,
102               const arma::colvec &inputs,
103               arma::colvec &outputs
104              );
105  void testNN(std::vector<std::vector<Neuron>> &network_,
106                const arma::mat &inpPattern,
107                const arma::mat &outPattern,
108                arma::colvec &errorVec
109        );
110 };
111
112  /*****NN operational functions*****/
113 void FeedForwardNetwork::applyWeights(const std::vector<double> &weights, const std::vector<
       double> &inputs, double &output) {
114   double dotProduct = 0.0;
115   for(int i=0; i<inputs.size(); i++) {
116     dotProduct = dotProduct + weights[i]*inputs[i];
117   }
118   output = dotProduct;
119 }
120
121 void FeedForwardNetwork::applyActivation(const double &input, double &output, std::string type)
       {
122   if(type.compare("logsig")==0) {
123     output = 1 / (1 + std::exp(-1*input));
124   } else if(type.compare("linear")==0) {
125     output = input;
126   } else {
127     std::cout << "Unsupported activation function chosen." << std::endl;
128   }
129 }
130
131 void FeedForwardNetwork::calculateActivationFnSlope(const double &input, double &output, std::
       string type) {
132   if(type.compare("logsig")==0) {
133     //sig'(x)=(1-sig(x))*sig(x)
134     output = (1.0-(1.0 / (1.0 + std::exp(-1.0*input))))*(1.0 / (1.0 + std::exp(-1.0*input)));
135   } else if(type.compare("linear")==0) {
136     //sig'(x) = 1
137     output = 1;
138   } else {
139     std::cout << "Unsupported activation function chosen." << std::endl;
140   }
141 }
142
143 void FeedForwardNetwork::applyLayer(std::vector<Neuron> &currentLayerNeurons, std::vector<Neuron
       > &nextLayerNeurons) {
144   for(int i=0; i<currentLayerNeurons.size(); i++) {
145     //apply weights
146     applyWeights(currentLayerNeurons[i].weights,
147             currentLayerNeurons[i].inputs,
148             currentLayerNeurons[i].netVal
149             );
150     //apply activation
151     applyActivation(currentLayerNeurons[i].netVal,
152              currentLayerNeurons[i].yVal,
153              currentLayerNeurons[i].activationType
154              );
155     //calculate activation function slope
156     calculateActivationFnSlope(currentLayerNeurons[i].netVal,
157                   currentLayerNeurons[i].actFnSlope,
158                   currentLayerNeurons[i].activationType
159                   );
160   }
161   //update inputs in next node
162   for(int i=0; i<nextLayerNeurons.size(); i++) {
163     for(int j=0; j<currentLayerNeurons.size(); j++) {
164       nextLayerNeurons[i].inputs[j] = currentLayerNeurons[j].yVal;
165     }
166   }
167 }
168
169 void FeedForwardNetwork::applyLayer(std::vector<Neuron> &currentLayerNeurons, arma::colvec &
       outputs) {
170   for(int i=0; i<currentLayerNeurons.size(); i++) {
171     //apply weights
172     applyWeights(currentLayerNeurons[i].weights,
173             currentLayerNeurons[i].inputs,
```

```
174             currentLayerNeurons[i].netVal
175             );
176     //apply activation
177     applyActivation(currentLayerNeurons[i].netVal,
178             currentLayerNeurons[i].yVal,
179             currentLayerNeurons[i].activationType
180             );
181     //calculate activation function slope
182     calculateActivationFnSlope(currentLayerNeurons[i].netVal,
183                     currentLayerNeurons[i].actFnSlope,
184                     currentLayerNeurons[i].activationType
185                     );
186
187   }
188   //update output vector
189   outputs.set_size(currentLayerNeurons.size());
190   for(int i=0; i<currentLayerNeurons.size(); i++) {
191     outputs(i) = currentLayerNeurons[i].yVal;
192   }
193 }
194
195 void FeedForwardNetwork::applyInputs(const arma::colvec &inputs, std::vector<Neuron> &
      firstLayerNeurons) {
196   for(int i=0; i<firstLayerNeurons.size(); i++) {
197     for(int j=0; j<inputs.n_elem; j++) {
198       firstLayerNeurons[i].inputs[j] = inputs(j);
199     }
200   }
201 }
202
203 void FeedForwardNetwork::initNeuronWeights(std::vector<double> &weights) {
204   for(int i=0; i<weights.size(); i++) {
205     // weights[i] = 2*mlpack::math::Random()-1; //scale [-1,1]
206     weights[i] = mlpack::math::ClampRange(mlpack::math::RandNormal(0,1),-1,1);//2/15);
207   }
208 }
209
210 void FeedForwardNetwork::reInitNetwork() {
211
212   for(int i=0; i<network_.size(); i++) {
213     for(int j=0; j<network_[i].size(); j++) {
214       //init weights
215       initNeuronWeights(network_[i][j].weights);
216     }
217
218   }
219
220 }
221
222 void FeedForwardNetwork::initNetwork( int nInputs, int nLayers,
223           std::vector<int> neuronsPerLayer,
224           std::vector<std::string> activationTypePerLayer
225           )
226 {
227   //resize network_
228   network_.resize(nLayers);
229
230   for(int i=0; i<network_.size(); i++) {
231     //resize layer
232     network_[i].resize(neuronsPerLayer[i]);
233
234     for(int j=0; j<network_[i].size(); j++) {
235       //resize neuron inputs/weights and init weights
236       if(i==0) {
237         network_[i][j].inputs.resize(nInputs);
238         network_[i][j].weights.resize(nInputs);
239       } else {
240         network_[i][j].inputs.resize(network_[i-1].size());
241         network_[i][j].weights.resize(network_[i-1].size());
242       }
243       //init weights
244       initNeuronWeights(network_[i][j].weights);
245       //init activation type
246       network_[i][j].activationType = activationTypePerLayer[i];
247       //resize delta vector size to number of outputs
248       network_[i][j].deltaVal.resize(neuronsPerLayer[neuronsPerLayer.size()-1]);
249     }
250
251   }
252
253 }
254
255
256 void FeedForwardNetwork::forwardPropagate(std::vector<std::vector<Neuron>> &network,
257             const arma::colvec &inputs,
258             arma::colvec &outputs
259           )
260 {
261   //apply inputs
262   applyInputs(inputs,network[0]);
263
264   //for each layer
```

```
265   for(int i=0; i<network.size(); i++) {
266     if(i != network.size()-1) {
267       applyLayer(network[i],network[i+1]);
268     } else {
269       applyLayer(network[i],outputs);
270     }
271   }
272 }
273
274 void FeedForwardNetwork::forwardPropagate( const arma::colvec &inputs, arma::colvec &outputs)
275 {
276   //apply inputs
277   applyInputs(inputs,network_[0]);
278
279   //for each layer
280   for(int i=0; i<network_.size(); i++) {
281     if(i != network_.size()-1) {
282       applyLayer(network_[i],network_[i+1]);
283     } else {
284       applyLayer(network_[i],outputs);
285     }
286   }
287 }
288
289 void FeedForwardNetwork::backwardPropagate(std::vector<std::vector<Neuron>> &network,
290             const arma::colvec &actualoutputVec,
291             const arma::colvec &correctOutputVec,
292             arma::colvec &errorVec
293             )
294 {
295   //output m
296   //layer k
297   //neuron j in layer k
298   //i-th input to neuron j
299
300   //resize to number of outputs
301   errorVec.set_size(actualoutputVec.size());
302
303   //iterate over all outputs
304   for(int m=0; m<network[network.size()-1].size(); m++) {
305     //calculate error
306     errorVec(m) = correctOutputVec(m)-actualoutputVec(m);
307
308     //iterate over all layers
309     for(int k=network.size()-1; k>=0; k--) {
310       //output layer
311       if(k==network.size()-1) {
312         //iterate on all neurons in layer
313         for(int j=0; j<network[k].size(); j++) {
314           if(j==m) { //if neuron j is the output m neuron
315             network[k][j].deltaVal[m] = network[k][j].actFnSlope;
316           } else { //all other neurons are zeroed
317             network[k][j].deltaVal[m] = 0;
318           }
319         }
320       }
321       //layer before output layer
322       else if(k==network.size()-2) {
323         //iterate on all neurons in layer
324         for(int j=0; j<network[k].size(); j++) {
325           //bp delta from inputs of (k+1)-th layer to
326           //outputs of k-th layer
327           network[k][j].deltaVal[m] =
328             network[k+1][m].deltaVal[m] *
329             network[k+1][m].weights[j];
330           //bp delta from outs of k-th layer to
331           //inputs of k-th layer
332           network[k][j].deltaVal[m] =
333             network[k][j].deltaVal[m] *
334             network[k][j].actFnSlope;
335         }
336       }
337       //all other layers
338       else {
339         //iterate on all neurons in layer
340         for(int j=0; j<network[k].size(); j++) {
341           double tmp=0;
342           //iterate on all inputs neuron j contibributed to
343           //in the next layer
344           for(int i=0; i<network[k+1].size(); i++) {
345             //bp delta from inputs of (k+1)-th layer to
346             //outputs of k-th layer
347             tmp = tmp +
348                  (network[k+1][i].deltaVal[m] *
349                   network[k+1][i].weights[j]);
350           }
351           network[k][j].deltaVal[m] = tmp;
352           //bp delta from outs of k-th layer to
353           //inputs of k-th layer
354           network[k][j].deltaVal[m] =
355             network[k][j].deltaVal[m] *
356             network[k][j].actFnSlope;
```

```
357         }
358       }
359     }
360   }
361 }
362
363
364
365 void FeedForwardNetwork::calculateJacobianMatrix(std::vector<std::vector<Neuron>> &network,
366                 arma::mat &jMat,
367                 arma::colvec &errorVec,
368                 arma::colvec &allCurrentWeights,
369                 const arma::mat &inpPattern,
370                 const arma::mat &outPattern
371                 )
372 {
373   arma::colvec fwdPropOutput;
374   arma::colvec onePatternErrorVec;
375   const int nOutputs = network[network.size()-1].size();
376
377   //resize jacobian matrix
378   int jMatCols=0;
379   for(int i=0; i<network.size(); i++) {
380     if(i==0) {
381       jMatCols = jMatCols + network[i].size()*network[i][0].inputs.size();
382     } else {
383       jMatCols = jMatCols + network[i-1].size()*network[i].size();
384     }
385   }
386
387   int jMatRows=nOutputs*inpPattern.n_cols;
388   jMat.set_size(jMatRows,jMatCols);
389   //resize error vector
390   errorVec.set_size(jMatRows);
391   //resize current weight vector
392   allCurrentWeights.set_size(jMatCols);
393
394   int colIt;
395   int weightIt=0;
396   for(int p=0; p<inpPattern.n_cols; p++) {
397     forwardPropagate(network,inpPattern.col(p),fwdPropOutput);
398     backwardPropagate(network,fwdPropOutput,outPattern.col(p),onePatternErrorVec);
399
400     //iterate over all outputs
401     for(int m=0; m<nOutputs; m++) {
402       //iterate over all layers
403       colIt=0;
404       for(int k=0; k<network.size(); k++) {
405         //iterate over all neurons in layer k
406         for(int j=0; j<network[k].size(); j++) {
407           //iterate over all inputs into neuron j
408           for(int i=0; i<network[k][j].inputs.size(); i++) {
409             //Jacobian Matrix
410             jMat(p*nOutputs+m, colIt) = -1*network[k][j].deltaVal[m]*network[k][j].inputs[i];
411             colIt++;
412
413             //create weight vector on first pass through
414             if(m==0 && p==0) {
415               allCurrentWeights(weightIt) = network[k][j].weights[i];
416               weightIt++;
417             }
418           }
419         }
420       }
421
422       //populate error vector
423       errorVec(p*nOutputs+m) = onePatternErrorVec(m);
424     }
425
426   }
427
428 }
429
430 void FeedForwardNetwork::calculateRecursiveWeights(const std::vector<std::vector<Neuron>> &
      network,
431                 const arma::colvec &allCurrentWeights,
432                 const arma::colvec &errorVec,
433                 const arma::colvec &grad,
434                 double mu,
435                 arma::colvec &allNewWeights
436                )
437 {
438   // #ifdef LOGGING
439   // logFile << "CALCULATERECURSIVEWEIGHTS flag 1" <<std::endl;
440   // #endif
441   allNewWeights.set_size(allCurrentWeights.n_elem);
442   //arma::mat IdMat;
443   //IdMat.eye(jMat.n_cols,jMat.n_cols);
444   //std::cout <<"here"<<std::endl;
445   //std::cout << jMat << std::endl;
446   //grad.zeros();
447   // #ifdef LOGGING
```

```
448   // logFile << "CALCULATERECURSIVEWEIGHTS flag 2, pMat:" <<rlmHelper_.pMat.n_rows<<","<<
        rlmHelper_.pMat.n_cols<<" grad: "<< grad.n_rows<<","<<grad.n_cols<<
449   //                         " Errorvec: "<<errorVec.n_rows<<","<<errorVec.n_cols <<std::endl;
450   // #endif
451
452   allNewWeights = allCurrentWeights + rlmHelper_.pMat * grad * arma::mean(errorVec);
453
454   // #ifdef LOGGING
455   // logFile << "CALCULATERECURSIVEWEIGHTS flag 3" <<std::endl;
456   // #endif
457   //std::cout <<"J'J : " << std::endl << ((double)(1.0/jMat.n_rows))*arma::trans(jMat)*jMat <<
        std::endl;
458   //std::cout <<"Je : " << std::endl << ((double)(1.0/jMat.n_rows))*arma::trans(jMat)*errorVec;
459 }
460
461
462 void FeedForwardNetwork::calculateWeightUpdate(const std::vector<std::vector<Neuron>> &network,
463                const arma::mat &jMat,
464                const arma::colvec &allCurrentWeights,
465                const arma::colvec &errorVec,
466                double mu,
467                arma::colvec &allNewWeights
468               )
469 {
470   // #ifdef LOGGING
471   //  auto start = boost::posix_time::microsec_clock::local_time();
472   // #endif
473
474   allNewWeights.set_size(allCurrentWeights.n_elem);
475   arma::mat IdMat;
476   IdMat.eye(jMat.n_cols,jMat.n_cols);
477   //std::cout <<"here"<<std::endl;
478   //std::cout << jMat << std::endl;
479   allNewWeights = allCurrentWeights - arma::inv(((double)(1.0/jMat.n_rows))*arma::trans(jMat)*
        jMat + mu*IdMat)*(((double)(1.0/jMat.n_rows))*arma::trans(jMat)*errorVec);
480   // #ifdef LOGGING
481   //  logFile << ":+:+:calcweightupdateTime: " << boost::posix_time::microsec_clock::local_time
        ()-start<<std::endl;
482   // // debugLogFile << "jmatCOls: " << jMat.n_cols<<std::endl;
483   // #endif
484   //std::cout <<"J'J : " << std::endl << ((double)(1.0/jMat.n_rows))*arma::trans(jMat)*jMat <<
        std::endl;
485   //std::cout <<"Je : " << std::endl << ((double)(1.0/jMat.n_rows))*arma::trans(jMat)*errorVec;
486 }
487
488 double FeedForwardNetwork::computeMeanSquareError(const arma::colvec &errorVec, int nOutputs,
        int nPatterns) {
489   double nOutputsDbl = (double) nOutputs;
490   double nPatternsDbl = (double) nPatterns;
491   arma::mat error = ((double) (1.0/(nOutputsDbl*nPatternsDbl)))*arma::trans(errorVec)*errorVec;
492   return error(0,0);
493 }
494
495 void FeedForwardNetwork::updateNetworkWithWeights(std::vector<std::vector<Neuron>> &network,
496                const arma::colvec &weights
497               )
498 {
499   int weightIt=0;
500   //iterate over layer k
501   for(int k=0; k<network.size(); k++) {
502     //iterate over all neurons in layer k
503     for(int j=0; j<network[k].size(); j++) {
504       //iterate over all inputs into neuron j
505       for(int i=0; i<network[k][j].inputs.size(); i++) {
506         network[k][j].weights[i] = weights(weightIt);
507         weightIt++;
508       }
509     }
510   }
511 }
512
513 void FeedForwardNetwork::testNN(std::vector<std::vector<Neuron>> &network_,
514                const arma::mat &inpPattern,
515                const arma::mat &outPattern,
516                arma::colvec &errorVec)
517 {
518   arma::colvec outputs;
519   arma::colvec singleRunErrorVec;
520
521   errorVec.set_size(outPattern.n_cols*outPattern.n_rows);
522
523   //iterate over all patterns
524   for(int p=0; p<outPattern.n_cols; p++) {
525     forwardPropagate(inpPattern.col(p), outputs);
526     //std::cout << "old: " << errorVec.subvec(p*outPattern.n_rows,(p+1)*outPattern.n_rows-1) <<
        std::endl;
527     errorVec.subvec(p*outPattern.n_rows,(p+1)*outPattern.n_rows-1) = outPattern.col(p) - outputs
        ;
528     //std::cout << "new: " << errorVec.subvec(p*outPattern.n_rows,(p+1)*outPattern.n_rows-1) <<
        std::endl;
529   }
530 }
```

```
531
532 void FeedForwardNetwork::runLM(const arma::mat &inpPatternTrain,
533                 const arma::mat &outPatternTrain,
534                 const arma::mat &inpPatternValid,
535                 const arma::mat &outPatternValid,
536                 double minError,
537                 double minGrad,
538                 double maxMu,
539                 int maxIterations,
540                 int maxValidationFails
541                  )
542 {
543   int m;
544   int loopIter;
545   arma::mat jMat;
546   arma::colvec errorVecForEval;
547   arma::colvec errorVecForJe;
548   arma::colvec allCurrentWeights;
549   arma::colvec allNewWeights;
550   double mu;
551   double lastError = arma::datum::inf;
552   double newError;
553   bool reCalc;
554   bool extremeMuReached;
555   arma::mat gradTmpMat;
556   double grad = arma::datum::inf;
557   double newErrorVal;
558   double lastErrorVal = arma::datum::inf;
559   double bestErrorVal = arma::datum::inf;
560   int valFailIter;
561   int maxCntLoop = 0;
562
563
564   //calculate jacobian
565   loopIter=0;
566   mu=.001;
567   valFailIter=0;
568   auto start = boost::posix_time::microsec_clock::local_time();
569   auto tmp = boost::posix_time::microsec_clock::local_time();
570   while(((lastError-minError)>1e-12) &&
571         (loopIter < maxIterations) &&
572         //((lastError-newError)>1e-12) &&
573         ((grad-minGrad)>1e-12) &&
574         (mu <= maxMu) &&
575         (valFailIter != maxValidationFails)
576     )
577   {
578     m = 0;
579     //calculate jacobian
580     //std::cout << "pre-jacobian"  << std::endl;
581
582     calculateJacobianMatrix(network_,jMat,errorVecForJe,allCurrentWeights,inpPatternTrain,
       outPatternTrain);
583
584     //std::cout << "post-jacobian" << std::endl;
585     errorVecForEval = errorVecForJe;
586     lastError = computeMeanSquareError(errorVecForEval,jMat.n_rows,1);
587
588     do {
589       // #ifdef LOGGING
590       //   start = boost::posix_time::microsec_clock::local_time();
591       // #endif
592
593       //update weights, evaluate, compare with current weight's performance
594       //std::cout << "calc weights" << std::endl;
595       // logFile<<"a1"<<allNewWeights <<std::endl;
596
597       calculateWeightUpdate(network_,jMat,allCurrentWeights,errorVecForJe,mu,allNewWeights);
598
599       // #ifdef LOGGING
600       // start = boost::posix_time::microsec_clock::local_time();
601       // // if(loopIter == 10)
602       // //   debugLogFile << ":++:weightUpdate Loop time :" <<boost::posix_time::microsec_clock
       ::local_time()- start <<std::endl;
603       // #endif
604
605       //logFile<<"a2"<<allNewWeights <<std::endl;
606       //std::cout << "update weights" << std::endl;
607       //std::cout << "new weights: " << allNewWeights << std::endl;
608       updateNetworkWithWeights(network_,allNewWeights);
609
610       //std::cout << "test nn" << std::endl;
611       testNN(network_,inpPatternTrain,outPatternTrain,errorVecForEval);
612       //std::cout << "errorVec: " << errorVecForEval << std::endl;
613       newError = computeMeanSquareError(errorVecForEval,jMat.n_rows,1);
614
615       //std::cout << loopIter << ":" << lastError << " " << newError << " " << (bool) (newError>
       lastError) << " " << mu << std::endl;
616
617       if((newError-lastError)>1e-12 /*|| m>0*/) {
618         //revert weights
619         updateNetworkWithWeights(network_,allCurrentWeights);
```

```
620
621           //need to use more grad descent
622           mu = mu*10.0;
623           if(mu>1e11) {
624             mu=1e11;
625           }
626
627         } else {
628           //can use more of newton's method
629           mu = mu/10.0;
630           if(mu<1e-300) {
631             mu=1e-300;
632           }
633         }
634         m=m+1;
635       //  #ifdef LOGGING
636       //   debugLogFile << "grad time: "<< boost::posix_time::microsec_clock::local_time() - start
      <<std::endl;
637       // #endif
638       // #ifdef LOGGING
639       // if(loopIter == 10)
640       //   debugLogFile << ":++:valFail Loop time :" <<boost::posix_time::microsec_clock::
      local_time()- start <<std::endl;
641       // #endif
642         // #ifdef LOGGING
643         // if(loopIter == 10)
644         //   logFile << ":++:Post weightUpdate Loop time :" <<boost::posix_time::microsec_clock::
      local_time()- start <<std::endl;
645         // #endif
646       }
647       while(((newError-lastError)>1e-12)) ;
648
649
650       if(m > maxCntLoop)
651         maxCntLoop = m;
652
653       //Calculate Grad for Iteration
654       calculateJacobianMatrix(network_,jMat,errorVecForJe,allNewWeights,inpPatternTrain,
      outPatternTrain);
655       //grad = 2*sqrt(1/(M*P)*Je.^2)
656
657       gradTmpMat = 2*arma::sqrt(arma::trans(((double)(1.0/jMat.n_rows))*arma::trans(jMat)*
      errorVecForJe)*
658           ((double)(1.0/jMat.n_rows))*arma::trans(jMat)*errorVecForJe));
659       grad = gradTmpMat(0,0);
660
661       //Calculate Validation Fail for Iteration and save best performance
662       testNN(network_,inpPatternValid,outPatternValid,errorVecForEval);
663       newErrorVal = computeMeanSquareError(errorVecForEval,outPatternValid.n_rows,outPatternValid.
      n_cols);
664       /*if(newErrorVal < lastErrorVal) {
665         valFailIter = 0;
666       } else { //validation fail
667         valFailIter++;
668       }
669       lastErrorVal = newErrorVal;
670       //Save "best performing" weights and save them
671       if(newErrorVal < bestErrorVal) {
672         trainedWeights_ = allNewWeights;
673         bestErrorVal = newErrorVal;
674       }*/
675       if(newErrorVal < bestErrorVal) {
676         valFailIter = 0; //reset val iterator
677         trainedWeights_ = allNewWeights; //save best weights
678         bestErrorVal = newErrorVal;
679       } else { //validation fail
680         valFailIter++;
681       }
682
683
684       //print progress
685       // #ifdef LOGGING
686       // logFile << loopIter << ":" << "grad: " << grad << ", perf: " << newErrorVal << ",
      valStops: " << valFailIter << std::endl;
687       // #endif
688       loopIter++;
689       // #ifdef LOGGING
690       // if (loopIter == 10)
691       //   logFile << ":+:loopIter loop time: " << boost::posix_time::microsec_clock::local_time()
      - start <<", valFailIter: " << valFailIter <<", loopIter:" << loopIter<<std::endl;
692       // #endif
693     }
694     // #ifdef LOGGING
695     //   logFile << "runLM finished: " << boost::posix_time::microsec_clock::local_time() - start
        <<", valFailIter: " << valFailIter <<", loopIter:" << loopIter<<std::endl;
696     // #endif
697     // #ifdef LOGGING
698     //   debugLogFile << "loopIter: "<<loopIter<<" valFailIter: "<<valFailIter << " maxCntLoop: "<<
        maxCntLoop <<std::endl;
699     // #endif
700     updateNetworkWithWeights(network_,trainedWeights_);
701   }
```

```
702
703 void FeedForwardNetwork::runRLM(
704                 const arma::mat &inpPatternTrain,
705                 const arma::mat &outPatternTrain,
706                 const arma::mat &inpPatternValid,
707                 const arma::mat &outPatternValid,
708
709                 double minError,
710                 double minGrad,
711                 double maxMu,
712                 int maxIterations,
713                 int maxValidationFails
714                  )
715 {
716   int m;
717   int loopIter;
718   arma::mat jMat;
719   arma::colvec errorVecForEval;
720   arma::colvec errorVecForJe;
721   arma::colvec allCurrentWeights;
722   arma::colvec tmpWeights;
723   arma::colvec allNewWeights;
724   double mu;
725   double lastError = arma::datum::inf;
726   double newError;
727   bool reCalc;
728   bool extremeMuReached;
729   arma::mat gradTmpMat;
730   arma::colvec grad ;
731   double gradPart = arma::datum::inf;
732   double newErrorVal;
733   double lastErrorVal = arma::datum::inf;
734   double bestErrorVal = arma::datum::inf;
735   int valFailIter;
736
737   // #ifdef LOGGING
738   // debugLogFile <<"RLM Flag 1" <<std::endl;
739   // #endif
740
741   //calculate jacobian
742   loopIter=0;
743   mu=.001;
744   valFailIter=0;
745   exportWeights(allCurrentWeights);
746
747   while(((lastError-minError)>1e-12) &&
748         (loopIter < maxIterations) &&
749         //((lastError-newError)>1e-12) &&
750         ((gradPart-minGrad)>1e-12) &&
751         (mu <= maxMu) &&
752         (valFailIter != maxValidationFails)
753       )
754   {
755     // #ifdef LOGGING
756     // if (loopIter % (maxIterations/4) == 1)
757     //   logFile << "loopIter:" << loopIter <<std::endl;
758     // #endif
759     // #ifdef LOGGING
760     // logFile <<"RLM Flag 2" <<std::endl;
761     // #endif
762
763     m = 0;
764     //calculate jacobian
765     //std::cout << "pre-jacobian"  << std::endl;
766     calculateJacobianMatrix(network_,jMat,errorVecForJe,allCurrentWeights,inpPatternTrain,
       outPatternTrain);
767     tmpWeights = allCurrentWeights;
768     // gradTmpMat = 2*arma::sqrt(arma::trans(((double)(1.0/jMat.n_rows))*arma::trans(jMat)*
       errorVecForJe)*
769     //         (((double)(1.0/jMat.n_rows))*arma::trans(jMat)*errorVecForJe));
770     /*****************************check if this is ok for grad
       ************************************************************/
771     do {
772       gradTmpMat = 2*arma::sqrt((((double)(1.0/jMat.n_rows))*arma::trans(jMat)*errorVecForJe) *
773                   arma::trans(((double)(1.0/jMat.n_rows))*arma::trans(jMat)*errorVecForJe));
774       grad = gradTmpMat.diag();
775       // #ifdef LOGGING
776       // logFile << "RLM FLag 2a, gradTmpMat: "<< gradTmpMat.size()<<std::endl;
777       // #endif
778
779       gradPart = gradTmpMat(0,0);
780       errorVecForEval = errorVecForJe;
781       lastError = computeMeanSquareError(errorVecForEval,jMat.n_rows,1);
782       int numTrainSamps = inpPatternTrain.n_rows;
783       // #ifdef LOGGING
784       //   logFile <<"RLM Flag 3: "<<grad.n_rows<<","<<grad.n_cols<<" numTrainSamps: "<<
       numTrainSamps <<std::endl;
785       // #endif
786
787       //*COPY TMPWEIGHTS TO ALL NEW WEIGHTS*//
788
789       for( int i = 0; i < numTrainSamps; i++){
```

```
790         // #ifdef LOGGING
791         //  logFile <<"RLM Flag 3a: " <<i<<std::endl;
792         // #endif
793         rlmHelper_.recursiveUpdate(network_,mu,errorVecForJe,grad,allCurrentWeights,
    inpPatternTrain,outPatternTrain); //* GOTTA DO THIS TO UPDATE ALL SAMPLES*//
794         // #ifdef LOGGING
795         //  logFile <<"RLM Flag 3b" <<std::endl;
796         // #endif
797         //Calculate Grad for Iteration
798         // #ifdef LOGGING
799         //  logFile <<"RLM Flag 3c" <<std::endl;
800         // #endif
801         calculateJacobianMatrix(network_,jMat,errorVecForJe,tmpWeights,inpPatternTrain,
    outPatternTrain);
802         //grad = 2*sqrt(1/(M*P)*Je.^2)
803         // #ifdef LOGGING
804         //  logFile <<"RLM Flag 3d" <<std::endl;
805         // #endif
806         gradTmpMat = 2*arma::sqrt((((double)(1.0/jMat.n_rows))*arma::trans(jMat)*errorVecForJe)
    *
807                 arma::trans(((double)(1.0/jMat.n_rows))*arma::trans(jMat)*errorVecForJe));
808
809         // gradTmpMat = 2*arma::sqrt(arma::trans(((double)(1.0/jMat.n_rows))*arma::trans(jMat)*
    errorVecForJe)*
810         //     ((double)(1.0/jMat.n_rows))*arma::trans(jMat)*errorVecForJe));
811         grad = gradTmpMat.diag();//gradTmpMat(0,0);
812         gradPart = gradTmpMat(0,0);
813         // logFile<<"a1"<<i<<" "<<allNewWeights <<std::endl;
814         calculateRecursiveWeights(network_,tmpWeights,errorVecForJe,grad,mu,allNewWeights);
815         // logFile<<"a2"<<allNewWeights <<std::endl;
816         updateNetworkWithWeights(network_,allNewWeights);
817         for (int j = 0; j < tmpWeights.size(); j++){
818         tmpWeights[j] = allNewWeights[j];
819         }
820
821       }
822
823       // #ifdef LOGGING
824       // logFile <<"RLM Flag 4, Mu: "<<mu <<std::endl;
825       // #endif
826
827       updateNetworkWithWeights(network_,allNewWeights);
828
829       //std::cout << "test nn" << std::endl;
830       testNN(network_,inpPatternTrain,outPatternTrain,errorVecForEval);
831       //std::cout << "errorVec: " << errorVecForEval << std::endl;
832       newError = computeMeanSquareError(errorVecForEval,jMat.n_rows,1);
833       //std::cout << loopIter << ":" << lastError << " " << newError << " " << (bool) (newError>
    lastError) << " " << mu << std::endl;
834
835       if((newError-lastError)>1e-12 /*|| m>0*/) {
836         //revert weights
837         // #ifdef LOGGING
838         // logFile << "New error:"<<newError<<std::endl<<"Last error: "<<lastError <<std::endl;
839         // #endif
840         updateNetworkWithWeights(network_,allCurrentWeights);
841         /*************RESET RECURSIVE HELPER TOO******************/
842         //need to use more grad descent
843         mu = mu*10.0;
844         if(mu>1e11) {
845           mu=1e11;
846         }
847
848       } else {
849         //can use more of newton's method
850         mu = mu/10.0;
851         if(mu<1e-30) {//0) {
852           mu=1e-30;//0;
853         }
854       }
855       m=m+1;
856     }
857     while(((newError-lastError)>1e-12)) ;
858     //print progress
859
860     //Calculate Grad for Iteration
861     //* GET GRAD HERE THE NORMAL WAY*//
862
863     calculateJacobianMatrix(network_,jMat,errorVecForJe,allNewWeights,inpPatternTrain,
    outPatternTrain);
864     //grad = 2*sqrt(1/(M*P)*Je.^2)
865     gradTmpMat = 2*arma::sqrt(arma::trans(((double)(1.0/jMat.n_rows))*arma::trans(jMat)*
    errorVecForJe)*
866         ((double)(1.0/jMat.n_rows))*arma::trans(jMat)*errorVecForJe));
867     grad = gradTmpMat(0,0);
868
869     //Calculate Validation Fail for Iteration and save best performance
870     testNN(network_,inpPatternValid,outPatternValid,errorVecForEval);
871     // #ifdef LOGGING
872     // logFile << "errorVec: " << errorVecForEval << std::endl;
873     // #endif
874     newErrorVal = computeMeanSquareError(errorVecForEval,outPatternValid.n_rows,outPatternValid.
```

```
      n_cols);
875     /*if(newErrorVal < lastErrorVal) {
876       valFailIter = 0;
877     } else { //validation fail
878       valFailIter++;
879     }
880     lastErrorVal = newErrorVal;
881     //Save "best performing" weights and save them
882     if(newErrorVal < bestErrorVal) {
883       trainedWeights_ = allNewWeights;
884       bestErrorVal = newErrorVal;
885     }*/
886     if(newErrorVal < bestErrorVal) {
887       valFailIter = 0; //reset val iterator
888       trainedWeights_ = allNewWeights; //save best weights
889       bestErrorVal = newErrorVal;
890     } else { //validation fail
891       // #ifdef LOGGING
892       //  logFile << "Validation fail!" <<std::endl;
893       // #endif
894       valFailIter++;
895     }
896
897     //print progress
898     //std::cout << loopIter << ":" << "grad: " << grad << ", perf: " << newError << ", valStops:
       " << valFailIter << std::endl;
899     // #ifdef LOGGING
900     // debugLogFile << loopIter << ":" << " perf: " << newError << ", valStops: " << valFailIter
       << std::endl;
901     // #endif
902     loopIter++;
903   }
904   // #ifdef LOGGING
905   //  logFile << "no more loopIter, loopIter:" << loopIter <<std::endl;
906   // #endif
907   updateNetworkWithWeights(network_,trainedWeights_);
908 }
909
910 void FeedForwardNetwork::exportWeights(arma::colvec &weights) {
911   int nWeights;
912
913   //get number of weights in network
914   nWeights=0;
915   for(int i=0; i<network_.size(); i++) {
916     for(int j=0; j<network_[i].size(); j++) {
917       for(int k=0; k<network_[i][j].weights.size(); k++) {
918         nWeights++;
919       }
920     }
921   }
922
923   weights.zeros(nWeights);
924
925   //reformat for MLPack
926   int it=0;
927   for(int i=0; i<network_.size(); i++) {
928     for(int k=0; k<network_[i][0].weights.size(); k++) {
929       for(int j=0; j<network_[i].size(); j++) {
930         weights(it) = network_[i][j].weights[k];
931         it++;
932       }
933     }
934   }
935 }
936
937 void FeedForwardNetwork::importWeights(const arma::colvec &weights) {
938   //weight format should be for MLPack
939   int it=0;
940   for(int i=0; i<network_.size(); i++) {
941     for(int k=0; k<network_[i][0].weights.size(); k++) {
942       for(int j=0; j<network_[i].size(); j++) {
943         network_[i][j].weights[k] = weights(it);
944         it++;
945       }
946     }
947   }
948 }
949
950 #endif
```

D.6 TrainingDataBuffer.cpp

```
#include <mlpack/core.hpp>

#include "/home/tim/Desktop/rlnn5/Logging.hpp"

#include <iostream>

#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <mlpack/prereqs.hpp>
#include <fstream>
#include <boost/archive/tmpdir.hpp>
#include <boost/serialization/base_object.hpp>
#include <boost/serialization/utility.hpp>
#include <boost/serialization/list.hpp>
#include <boost/serialization/assume_abstract.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/serialization/serialization.hpp>
#include <algorithm>
#include <cmath>

using namespace mlpack;

namespace boost {
namespace serialization {
class access;
}
}

class TrainingDataBuffer {
  public:
    bool addTrainingSample(int actionID, const arma::colvec & outVec);
    void buildTrainingSet(  int nnIdx,
                const arma::mat & actionList,
                const std::vector<int> &inpBuffIdxs, const std::vector<int> &
    inpBuffExpandActions,
                const arma::mat &inpNormParams,
                const std::vector<int> &outBuffIdxs, const std::vector<int> &
    outBuffExpandActions,
                const arma::mat &outNormParams);

    bool pruneDataBuffer(double fracToKeep);

    void printBuffActionIDTime();
    void printBuffOutputVec();
    void printTrainTestInput();
    void printTrainTestOutput();

    std::vector<arma::mat> * getTrainTestInput();
    std::vector<arma::mat> * getTrainTestOutput();

    int getBufCntr();
    int getBufferSize();
    void resetBuffer();

  void buildTrainingColumn(   int nnIdx,
                const arma::mat & actionList,
                const std::vector<int> &inpBuffIdxs, const std::vector<int> &
    inpBuffExpandActions,
                const arma::mat &inpNormParams,
                const std::vector<int> &outBuffIdxs, const std::vector<int> &
    outBuffExpandActions,
                const arma::mat &outNormParams,
                int paramToSortBy,
                int ascendingOrderIdx,
                arma::colvec &trainingInCol, arma::colvec &trainingOutCol);

    TrainingDataBuffer(int nOutVecFeatures, int nTrainTestSamples, int nNeuralNets); //
    constructor

    struct InpOutBuffParams {
      std::vector<int> inpBuffIdxs;
      std::vector<int> inpBuffExpandActions;
      arma::mat inpNormParams;
      std::vector<int> outBuffIdxs;
      std::vector<int> outBuffExpandActions;
      arma::mat outNormParams;
    };

    void saveCurrentRun(std::string filename) {

      std::ofstream ofs(filename);
      //save data to archvie
      boost::archive::text_oarchive oa(ofs);
      //write class instance to archive
      oa & *this;
      //archive and stream closed with destructors are called
    };

```

```
84      void loadOldRun(std::string filename) {
85        std::ifstream ifs(filename);
86        boost::archive::text_iarchive ia(ifs);
87        //read class state from archive
88        ia & *this;
89        //archive and stream closed with destructors are called.
90      }
91
92    private:
93      arma::Mat<int> _buffActionIDTime;
94      arma::mat _buffOutputVec;
95
96      std::vector<arma::mat> _trainTestInput;
97      std::vector<arma::mat> _trainTestOutput;
98
99      double _bufCntr;
100
101     int _lastActionID;
102
103     friend class boost::serialization::access;
104     template<class Archive>
105     void serialize(Archive & ar, const int version) {
106       ar & _buffActionIDTime;
107       ar & _buffOutputVec;
108       ar & _trainTestInput;
109       ar & _trainTestOutput;
110       ar & _bufCntr;
111       ar & _lastActionID;
112     }
113 };
114
115 TrainingDataBuffer::TrainingDataBuffer(int nOutVecFeatures, int nTrainTestSamples, int
        nNeuralNets) {
116   _buffActionIDTime.set_size(2, nTrainTestSamples); //{actionID; lastModified}
117   _buffActionIDTime.fill(-1);
118   _buffOutputVec.set_size(nOutVecFeatures, nTrainTestSamples);
119
120   _trainTestInput.resize(nNeuralNets);
121   _trainTestOutput.resize(nNeuralNets);
122
123   _bufCntr = 0;
124 }
125
126 std::vector<arma::mat> * TrainingDataBuffer::getTrainTestInput() {
127   return &_trainTestInput;
128 }
129
130 std::vector<arma::mat> * TrainingDataBuffer::getTrainTestOutput() {
131   return &_trainTestOutput;
132 }
133
134 int TrainingDataBuffer::getBufCntr() {
135   return _bufCntr;
136 }
137
138 int TrainingDataBuffer::getBufferSize() {
139   return _buffActionIDTime.n_cols;
140 }
141
142 void TrainingDataBuffer::resetBuffer() {
143   _buffActionIDTime.fill(-1);
144   _bufCntr = 0;
145 }
146
147 void TrainingDataBuffer::printBuffActionIDTime() {
148   std::cout << _buffActionIDTime <<  std::endl;
149 }
150
151 void TrainingDataBuffer::printBuffOutputVec() {
152   std::cout << _buffOutputVec <<  std::endl;
153 }
154
155 void TrainingDataBuffer::printTrainTestInput() {
156   for(int i=0; i<_trainTestInput.size(); i++) {
157     std::cout << "NN_" << i << ":" << std::endl;
158     std::cout << _trainTestInput[i] <<  std::endl;
159   }
160 }
161
162 void TrainingDataBuffer::printTrainTestOutput() {
163   for(int i=0; i<_trainTestOutput.size(); i++) {
164     std::cout << "NN_" << i << ":" << std::endl;
165     std::cout << _trainTestOutput[i] <<  std::endl;
166   }
167 }
168
169 bool TrainingDataBuffer::addTrainingSample(int actionID, const arma::colvec & outVec) {
170   //search for action ID
171   arma::uvec currActionIdx = find(_buffActionIDTime.row(0)==actionID);
172   if(currActionIdx.is_empty()) { //action not in buffer
173     if(_bufCntr<_buffActionIDTime.n_cols) { //buffer not full yet
174       //add to next free spot
```

```
175       arma::uvec freeIdx = find(_buffActionIDTime.row(0)==-1,1,"first");
176       _buffActionIDTime(0,freeIdx(0)) = actionID;
177       _buffActionIDTime(1,freeIdx(0)) = 0;
178       _buffOutputVec.col(freeIdx(0)) = outVec;
179       _buffActionIDTime.row(1) = _buffActionIDTime.row(1) + arma::ones<arma::Row<unsigned int>>(
      _buffActionIDTime.n_cols);
180       _bufCntr++;
181     } else { //buffer full
182       //replace oldest entry
183       int oldestIdx = _buffActionIDTime.row(1).index_max();
184       _buffActionIDTime(0,oldestIdx) = actionID;
185       _buffActionIDTime(1,oldestIdx) = 0;
186       _buffOutputVec.col(oldestIdx) = outVec;
187       _buffActionIDTime.row(1) = _buffActionIDTime.row(1) + arma::ones<arma::Row<unsigned int>>(
      _buffActionIDTime.n_cols);
188     }
189   } else { //action is in buffer
190     //only add if last action is different than current action
191     if(_lastActionID != actionID) {
192       arma::uvec newerTimestamps = find(_buffActionIDTime.row(1) < _buffActionIDTime(1,
      currActionIdx(0)));
193       arma::Row<unsigned int> onesVec = arma::ones<arma::Row<unsigned int>>(newerTimestamps.
      n_elem);
194       //add 1 to all newer timestamps than the found Action IDX
195       _buffActionIDTime(onesVec,newerTimestamps) = _buffActionIDTime(onesVec,newerTimestamps) +
      1;
196       //add new timestamp
197       _buffActionIDTime(1,currActionIdx(0)) = 1;
198       //replace with new entry
199       _buffOutputVec.col(currActionIdx(0)) = outVec;
200       //_buffActionIDTime.row(1) = _buffActionIDTime.row(1) + arma::ones<arma::Row<unsigned int
      >>(_buffActionIDTime.n_cols);
201     }
202   }
203
204   _lastActionID = actionID;
205
206   return _bufCntr == _buffActionIDTime.n_cols;
207 }
208
209 bool TrainingDataBuffer::pruneDataBuffer(double fracToKeep) {
210   int nSampToKeep = (int) floor(fracToKeep*_buffActionIDTime.n_cols);
211   for(int i=0; i<_buffActionIDTime.n_cols; i++) {
212     if(_buffActionIDTime(1,i) > nSampToKeep) { //vals start at 1
213       _buffActionIDTime(0,i) = -1;
214       _buffActionIDTime(1,i) = 0;
215       _buffOutputVec.col(i) = -1.0*arma::ones<arma::vec>(_buffOutputVec.n_rows);
216     }
217   }
218
219   _bufCntr = nSampToKeep;
220   return _bufCntr == _buffActionIDTime.n_cols;
221 }
222
223 void TrainingDataBuffer::buildTrainingSet( int nnIdx, //index of NN's buffer to build (starts at
      0)
224                 const arma::mat & actionList, //n_action_types x n_permutations
225                 const std::vector<int> &inpBuffIdxs,  //idx -1 is actionID, idx0+ correspond to
      outVec passed into
226                                       //  addTrainingSample
227                 const std::vector<int> &inpBuffExpandActions, //same length as inpBuffIdxs. true
       if actionID should be
228                                       // expanded at this idx. False otherwise
229                 const arma::mat &inpNormParams, //cols: [param min, param max, new range min,
      new range max].
230                                 //rows: corresponds to idxs in inpBuffIdxs
231                 const std::vector<int> &outBuffIdxs,  //idx -1 is actionID, idx0+ correspond to
      outVec passed into
232                                       //  addTrainingSample
233                 const std::vector<int> &outBuffExpandActions, //same length as inpBuffIdxs. true
       if actionID should be
234                                       // expanded at this idx. False otherwise
235                 const arma::mat &outNormParams) //cols: [param min, param max, new range min,
      new range max].
236                                 //rows: corresponds to idxs in inpBuffIdxs
237 {
238   //determine number of rows of _trainTestInput
239   int nRows = 0;
240   for(int i=0; i<inpBuffExpandActions.size(); i++) {
241     if(inpBuffExpandActions[i]==1 || inpBuffExpandActions[i]==2) {
242       nRows=nRows+actionList.n_rows; //we're going to be expanding the action list
243     } else {
244       nRows++;  //outVec parameter
245     }
246   }
247   _trainTestInput[nnIdx].set_size(nRows,_buffActionIDTime.n_cols);
248
249   //start populating the input training buffer
250   for(int j=0; j<_trainTestInput[nnIdx].n_cols; j++) { //loop through columns of input training
       buffer (nTrainingSamples)
251     int rowIdx=0; //actual row idx of _trainTestInput (accounts for actionList expanding)
252     for(int k=0; k<inpBuffIdxs.size(); k++) { //loop through which idxs of features you want in
```

```
    the input
253       if(inpBuffExpandActions[k] == 1) { //if we're going to expand an action ID into its vector
      actions and scale from global action min/max
254         for(int m=0; m<actionList.n_rows; m++) { //loop through each element in a vector action
255           //normalize value and copy into
256           //val_norm = (new_range_max-new_range_min)*(val-param_min)/(param_max-param_min) +
    new_range_min
257           _trainTestInput[nnIdx](rowIdx,j) = (inpNormParams(k,3)-inpNormParams(k,2))*
258                             (actionList(m,_buffActionIDTime(0,j))-inpNormParams(k,0))/
259                             (inpNormParams(k,1)-inpNormParams(k,0))
260                             + inpNormParams(k,2);
261           rowIdx++;
262         }
263       } else if(inpBuffExpandActions[k] == 2) { //expand and scale from action's min/max
264         for(int m=0; m<actionList.n_rows; m++) { //loop through each element in a vector action
265           //normalize value and copy into
266           //val_norm = (new_range_max-new_range_min)*(val-param_min)/(param_max-param_min) +
    new_range_min
267           if((actionList.row(m).max()-actionList.row(m).min())/std::max(std::abs(actionList.row(
    m).max()),std::abs(actionList.row(m).min())) > 0.00001) {
268             _trainTestInput[nnIdx](rowIdx,j) = (inpNormParams(k,3)-inpNormParams(k,2))*
269                               (actionList(m,_buffActionIDTime(0,j))-actionList.row(m).min())/
270                               (actionList.row(m).max()-actionList.row(m).min())
271                               + inpNormParams(k,2);
272           } else {
273             _trainTestInput[nnIdx](rowIdx,j) = inpNormParams(k,3);
274           }
275           rowIdx++;
276         }
277       } else { //we're adding either a value from outVec that was added or the action ID #
    directly
278         if(inpBuffIdxs[k] != -1) {  //adding outVec value
279           _trainTestInput[nnIdx](rowIdx,j) = (inpNormParams(k,3)-inpNormParams(k,2))*
280                               (_buffOutputVec(inpBuffIdxs[k],j)-inpNormParams(k,0))/
281                               (inpNormParams(k,1)-inpNormParams(k,0))
282                               + inpNormParams(k,2);
283         } else { //add actionID directly (not sure why'd you want to ever do this, but i'm
    giving you the option)
284           _trainTestInput[nnIdx](rowIdx,j) = (inpNormParams(k,3)-inpNormParams(k,2))*
285                             (_buffActionIDTime(0,j)-inpNormParams(k,0))/
286                             (inpNormParams(k,1)-inpNormParams(k,0))
287                             + inpNormParams(k,2);
288         }
289         rowIdx++;
290       }
291     }
292   }
293
294   //determine number of rows of _trainTestOutput
295   nRows = 0;
296   for(int i=0; i<outBuffExpandActions.size(); i++) {
297     if(outBuffExpandActions[i]==1 || outBuffExpandActions[i]==2) {
298       nRows=nRows+actionList.n_rows; //we're going to be expanding the action list
299     } else {
300       nRows++;  //outVec parameter
301     }
302   }
303   _trainTestOutput[nnIdx].set_size(nRows,_buffActionIDTime.n_cols);
304
305   //start populating the output training buffer
306   for(int j=0; j<_trainTestOutput[nnIdx].n_cols; j++) { //loop through columns of output
    training buffer (nTrainingSamples)
307     int rowIdx=0; //actual row idx of _trainTestOutput (accounts for actionList expanding)
308     for(int k=0; k<outBuffIdxs.size(); k++) { //loop through which idxs of features you want in
    the output
309       if(outBuffExpandActions[k] == 1) { //if we're going to expand an action ID into its vector
      actions and scale from global min/max
310         for(int m=0; m<actionList.n_rows; m++) { //loop through each element in a vector action
311           //val_norm = (new_range_max-new_range_min)*(val-param_min)/(param_max-param_min) +
    new_range_min
312           _trainTestOutput[nnIdx](rowIdx,j) = (outNormParams(k,3)-outNormParams(k,2))*
313                             (actionList(m,_buffActionIDTime(0,j))-outNormParams(k,0))/
314                             (outNormParams(k,1)-outNormParams(k,0))
315                             + outNormParams(k,2);
316           rowIdx++;
317         }
318       } else if(outBuffExpandActions[k] == 2) { //expand and scale from action's min/max
319         for(int m=0; m<actionList.n_rows; m++) { //loop through each element in a vector action
320           //val_norm = (new_range_max-new_range_min)*(val-param_min)/(param_max-param_min) +
    new_range_min
321           double rowMax = arma::max(actionList.row(m));
322           double rowMin = arma::min(actionList.row(m));
323           if((rowMax-rowMin)/std::max(std::abs(rowMax),std::abs(rowMin)) > 0.00001) {
324             _trainTestOutput[nnIdx](rowIdx,j) = (outNormParams(k,3)-outNormParams(k,2))*
325                               (actionList(m,_buffActionIDTime(0,j))-rowMin)/
326                               (rowMax-rowMin)
327                               + outNormParams(k,2);
328           } else {
329             _trainTestOutput[nnIdx](rowIdx,j) = outNormParams(k,3);
330           }
331           rowIdx++;
332         }
```

```
333       } else { //we're adding either a value from outVec that was added or the action ID #
      directly
334         if(outBuffIdxs[k] != -1) {  //adding outVec value
335           _trainTestOutput[nnIdx](rowIdx,j) = (outNormParams(k,3)-outNormParams(k,2))*
336                             (_buffOutputVec(outBuffIdxs[k],j)-outNormParams(k,0))/
337                             (outNormParams(k,1)-outNormParams(k,0))
338                             + outNormParams(k,2);
339         } else { //add actionID directly (not sure why'd you want to ever do this, but i'm
      giving you the option)
340           _trainTestOutput[nnIdx](rowIdx,j) = (outNormParams(k,3)-outNormParams(k,2))*
341                            (_buffActionIDTime(0,j)-outNormParams(k,0))/
342                            (outNormParams(k,1)-outNormParams(k,0))
343                            + outNormParams(k,2);
344         }
345         rowIdx++;
346       }
347     }
348   }
349
350 }
351
352 void TrainingDataBuffer::buildTrainingColumn( int nnIdx, //index of NN's buffer to build (starts
       at 0)
353                 const arma::mat & actionList, //n_action_types x n_permutations
354                 const std::vector<int> &inpBuffIdxs,  //idx -1 is actionID, idx0+ correspond to
      outVec passed into
355                                       //  addTrainingSample
356                 const std::vector<int> &inpBuffExpandActions, //same length as inpBuffIdxs. true
       if actionID should be
357                                       // expanded at this idx. False otherwise
358                 const arma::mat &inpNormParams, //cols: [param min, param max, new range min,
      new range max].
359                                   //rows: corresponds to idxs in inpBuffIdxs
360                 const std::vector<int> &outBuffIdxs,  //idx -1 is actionID, idx0+ correspond to
      outVec passed into
361                                       //  addTrainingSample
362                 const std::vector<int> &outBuffExpandActions, //same length as inpBuffIdxs. true
       if actionID should be
363                                       // expanded at this idx. False otherwise
364                 const arma::mat &outNormParams, //cols: [param min, param max, new range min,
      new range max].
365                                   //rows: corresponds to idxs in inpBuffIdxs
366                 int paramToSortBy,          //-1: time, 0->size(outvec): param in outvec
367                 int ascendingOrderIdx,       //column number from buffer idx'd "sorted" by value
      in idx in ascending order
368                 arma::colvec &trainingInCol,  //input training vector
369                 arma::colvec &trainingOutCol) //output training vector
370 {
371   //determine number of rows of trainingCol
372   int nRows = 0;
373   for(int i=0; i<inpBuffExpandActions.size(); i++) {
374     if(inpBuffExpandActions[i]==1 || inpBuffExpandActions[i]==2) {
375       nRows=nRows+actionList.n_rows; //we're going to be expanding the action list
376     } else {
377       nRows++;  //outVec parameter
378     }
379   }
380   trainingInCol.set_size(nRows);
381
382   //find column we want to build
383   //std::cout << "HERE2A" << std::endl;
384   arma::uvec sortedIdxs;
385   if(paramToSortBy==-1) {
386     sortedIdxs = arma::sort_index(_buffActionIDTime.row(1),"ascend");
387   } else {
388     sortedIdxs = arma::sort_index(_buffOutputVec.row(paramToSortBy),"ascend");
389   }
390   //std::cout << "HERE2B" << std::endl;
391   int j = (int) sortedIdxs(ascendingOrderIdx);
392   //std::cout << "j: " << j << std::endl;
393   //std::cout << "ascendOrderIdx" << ascendingOrderIdx << std::endl;
394
395   //start populating the input training column
396   int rowIdx=0; //actual row idx of _trainTestInput (accounts for actionList expanding)
397   for(int k=0; k<inpBuffIdxs.size(); k++) { //loop through which idxs of features you want in
      the input
398     if(inpBuffExpandActions[k] == 1) { //if we're going to expand an action ID into its vector
      actions and scale from global action min/max
399       for(int m=0; m<actionList.n_rows; m++) { //loop through each element in a vector action
400         //normalize value and copy into
401         //val_norm = (new_range_max-new_range_min)*(val-param_min)/(param_max-param_min) +
      new_range_min
402         trainingInCol(rowIdx) = (inpNormParams(k,3)-inpNormParams(k,2))*
403                           (actionList(m,_buffActionIDTime(0,j))-inpNormParams(k,0))/
404                           (inpNormParams(k,1)-inpNormParams(k,0))
405                           + inpNormParams(k,2);
406         rowIdx++;
407       }
408     } else if(inpBuffExpandActions[k] == 2) { //expand and scale from action's min/max
409       for(int m=0; m<actionList.n_rows; m++) { //loop through each element in a vector action
410         //normalize value and copy into
411         //val_norm = (new_range_max-new_range_min)*(val-param_min)/(param_max-param_min) +
```

```
    new_range_min
412         if((actionList.row(m).max()-actionList.row(m).min())/std::max(std::abs(actionList.row(m)
    .max()),std::abs(actionList.row(m).min())) > 0.00001) {
413           trainingInCol(rowIdx) = (inpNormParams(k,3)-inpNormParams(k,2))*
414                                   (actionList(m,_buffActionIDTime(0,j))-actionList.row(m).min())/
415                                   (actionList.row(m).max()-actionList.row(m).min())
416                                   + inpNormParams(k,2);
417         } else {
418           trainingInCol(rowIdx) = inpNormParams(k,3);
419         }
420         rowIdx++;
421       }
422     } else { //we're adding either a value from outVec that was added or the action ID #
    directly
423       if(inpBuffIdxs[k] != -1) {  //adding outVec value
424         //std::cout << "HERE2C" << std::endl;
425         trainingInCol(rowIdx) = (inpNormParams(k,3)-inpNormParams(k,2))*
426                                 (_buffOutputVec(inpBuffIdxs[k],j)-inpNormParams(k,0))/
427                                 (inpNormParams(k,1)-inpNormParams(k,0))
428                                 + inpNormParams(k,2);
429         //std::cout << inpNormParams.row(k) << " " << _buffOutputVec(inpBuffIdxs[k],j) << std::
    endl;
430         //std::cout << "trainingInCol(i): " << trainingInCol(rowIdx) << std::endl;
431       } else { //add actionID directly (not sure why'd you want to ever do this, but i'm giving
    you the option)
432         trainingInCol(rowIdx) = (inpNormParams(k,3)-inpNormParams(k,2))*
433                                 (_buffActionIDTime(0,j)-inpNormParams(k,0))/
434                                 (inpNormParams(k,1)-inpNormParams(k,0))
435                                 + inpNormParams(k,2);
436       }
437       rowIdx++;
438     }
439   }
440   //std::cout << "HERE2D" << std::endl;
441   //determine number of rows of trainingOutCol
442   nRows = 0;
443   for(int i=0; i<outBuffExpandActions.size(); i++) {
444     if(outBuffExpandActions[i]==1 || outBuffExpandActions[i]==2) {
445       nRows=nRows+actionList.n_rows; //we're going to be expanding the action list
446     } else {
447       nRows++;  //outVec parameter
448     }
449   }
450   trainingOutCol.set_size(nRows);
451   //std::cout << "HERE2E" << std::endl;
452   //std::cout << trainingOutCol.n_elem << std::endl;
453
454   //start populating the output training buffer
455   rowIdx=0; //actual row idx of trainingOutCol (accounts for actionList expanding)
456   for(int k=0; k<outBuffIdxs.size(); k++) { //loop through which idxs of features you want in
    the output
457     if(outBuffExpandActions[k] == 1) { //if we're going to expand an action ID into its vector
    actions and scale from global min/max
458       for(int m=0; m<actionList.n_rows; m++) { //loop through each element in a vector action
459         //val_norm = (new_range_max-new_range_min)*(val-param_min)/(param_max-param_min) +
    new_range_min
460         trainingOutCol(rowIdx) = (outNormParams(k,3)-outNormParams(k,2))*
461                                  (actionList(m,_buffActionIDTime(0,j))-outNormParams(k,0))/
462                                  (outNormParams(k,1)-outNormParams(k,0))
463                                  + outNormParams(k,2);
464         rowIdx++;
465       }
466     } else if(outBuffExpandActions[k] == 2) { //expand and scale from action's min/max
467       for(int m=0; m<actionList.n_rows; m++) { //loop through each element in a vector action
468         //val_norm = (new_range_max-new_range_min)*(val-param_min)/(param_max-param_min) +
    new_range_min
469         double rowMax = arma::max(actionList.row(m));
470         double rowMin = arma::min(actionList.row(m));
471         if((rowMax-rowMin)/std::max(std::abs(rowMax),std::abs(rowMin)) > 0.00001) {
472           trainingOutCol(rowIdx) = (outNormParams(k,3)-outNormParams(k,2))*
473                                    (actionList(m,_buffActionIDTime(0,j))-rowMin)/
474                                    (rowMax-rowMin)
475                                    + outNormParams(k,2);
476         } else {
477           trainingOutCol(rowIdx) = outNormParams(k,3);
478         }
479         rowIdx++;
480       }
481     } else { //we're adding either a value from outVec that was added or the action ID #
    directly
482       if(outBuffIdxs[k] != -1) {  //adding outVec value
483         trainingOutCol(rowIdx) = (outNormParams(k,3)-outNormParams(k,2))*
484                                  (_buffOutputVec(outBuffIdxs[k],j)-outNormParams(k,0))/
485                                  (outNormParams(k,1)-outNormParams(k,0))
486                                  + outNormParams(k,2);
487       } else { //add actionID directly (not sure why'd you want to ever do this, but i'm giving
    you the option)
488         trainingOutCol(rowIdx) = (outNormParams(k,3)-outNormParams(k,2))*
489                                  (_buffActionIDTime(0,j)-outNormParams(k,0))/
490                                  (outNormParams(k,1)-outNormParams(k,0))
491                                  + outNormParams(k,2);
492       }
```

```
493         rowIdx++;
494       }
495     }
496
497     //std::cout << "HERE2F" << std::endl;
498
499 }
500
501 /*
502 int main() {
503     const int N_ACTIONS = 20;
504     const int N_OUTPUTS = 2;
505     const int N_SAMPLES = 10;
506     const int N_NNs = 2;
507     TrainingDataBuffer buff1(N_OUTPUTS, N_SAMPLES, N_NNs);
508
509     std::vector<int> inpBuffIdxs0 = {-1,0};
510     std::vector<int> inpBuffExpandActions0 = {1, 0};
511     arma::mat inpNormParams0 = { {0,(N_ACTIONS-1)*2.5,-1,1},
512                                  {0,1,-1,1}
513                                  };
514     std::vector<int> inpBuffIdxs1 = {0,1,-1};
515     std::vector<int> inpBuffExpandActions1 = {0, 0, 0};
516     arma::mat inpNormParams1 = { {0,1,0,2},
517                                  {0,1,0,2},
518                                  {0,N_ACTIONS-1,0,2}
519                                  };
520
521     arma::mat actionList(4,N_ACTIONS);
522
523     for(int i=0; i<actionList.n_cols; i++) {
524       actionList(0,i) = i*1;
525       actionList(1,i) = i*1.5;
526       actionList(2,i) = i*2;
527       actionList(3,i) = i*2.5;
528     }
529     //add until full
530     bool full = false;
531     int actionID;
532     arma::colvec outVec;
533     while(!full) {
534       int actionID = math::RandInt(0,N_ACTIONS);
535       arma::colvec outVec = math::Random()*arma::ones<arma::colvec>(N_OUTPUTS);
536       outVec(0)=outVec(0)*math::Random();
537       std::cout << "actionID: " << actionID << std::endl;
538       std::cout << "outVec:" << std::endl << outVec << std::endl;
539       full = buff1.addTrainingSample(actionID, outVec);
540       std::cout << "buffActionIDTime:" << std::endl;
541       buff1.printBuffActionIDTime();
542       std::cout << "buffOutputVec:" << std::endl;
543       buff1.printBuffOutputVec();
544     };
545     std::cout << "buffActionIDTime:" << std::endl;
546     buff1.printBuffActionIDTime();
547     std::cout << "buffOutputVec:" << std::endl;
548     buff1.printBuffOutputVec();
549
550     //add a couple more to overwrite buffer
551     for(int i=0; i<2; i++) {
552       int actionID = math::RandInt(0,N_ACTIONS);
553       arma::colvec outVec = math::Random()*arma::ones<arma::colvec>(N_OUTPUTS);
554       outVec(0)=outVec(0)*math::Random();
555       std::cout << "actionID: " << actionID << std::endl;
556       std::cout << "outVec:" << std::endl << outVec << std::endl;
557       full = buff1.addTrainingSample(actionID, outVec);
558       std::cout << "buffActionIDTime:" << std::endl;
559       buff1.printBuffActionIDTime();
560       std::cout << "buffOutputVec:" << std::endl;
561       buff1.printBuffOutputVec();
562     }
563
564     //add a couple of the same
565     for(int i=0; i<2; i++) {
566       int actionID = 0;
567       arma::colvec outVec = i*arma::zeros<arma::colvec>(N_OUTPUTS);
568       outVec(0)=outVec(0)*math::Random();
569       std::cout << "actionID: " << actionID << std::endl;
570       std::cout << "outVec:" << std::endl << outVec << std::endl;
571       full = buff1.addTrainingSample(actionID, outVec);
572       std::cout << "buffActionIDTime:" << std::endl;
573       buff1.printBuffActionIDTime();
574       std::cout << "buffOutputVec:" << std::endl;
575       buff1.printBuffOutputVec();
576     }
577
578     buff1.buildTrainingSet(0,actionList,
579                            inpBuffIdxs0,inpBuffExpandActions0,inpNormParams0,
580                            inpBuffIdxs1,inpBuffExpandActions1,inpNormParams1);
581     buff1.buildTrainingSet(1,actionList,
582                            inpBuffIdxs1,inpBuffExpandActions1,inpNormParams1,
583                            inpBuffIdxs0,inpBuffExpandActions0,inpNormParams0);
584     std::cout << "Action List:" << std::endl <<actionList <<  std::endl;
```

```
585   std::cout << "buffActionIDTime:" << std::endl;
586   buff1.printBuffActionIDTime();
587   std::cout << "buffOutputVec:" << std::endl;
588   buff1.printBuffOutputVec();
589   std::cout << "_trainTestInput:" << std::endl;
590   buff1.printTrainTestInput();
591   std::cout << "_trainTestOutput:" << std::endl;
592   buff1.printTrainTestOutput();
593
594   full = buff1.pruneDataBuffer(0.5);
595   std::cout << "buffActionIDTime:" << std::endl;
596   buff1.printBuffActionIDTime();
597   std::cout << "buffOutputVec:" << std::endl;
598   buff1.printBuffOutputVec();
599   std::cout << "_trainTestInput:" << std::endl;
600   buff1.printTrainTestInput();
601   std::cout << "_trainTestOutput:" << std::endl;
602   buff1.printTrainTestOutput();
603
604   while(!full) {
605     int actionID = math::RandInt(0,N_ACTIONS);
606     arma::colvec outVec = math::Random()*arma::ones<arma::colvec>(N_OUTPUTS);
607     outVec(0)=outVec(0)*math::Random();
608     std::cout << "actionID: " << actionID << std::endl;
609     std::cout << "outVec:" << std::endl << outVec << std::endl;
610     full = buff1.addTrainingSample(actionID, outVec);
611     std::cout << "buffActionIDTime:" << std::endl;
612     buff1.printBuffActionIDTime();
613     std::cout << "buffOutputVec:" << std::endl;
614     buff1.printBuffOutputVec();
615   }
616
617 }
618 */
```

D.7 ApplicationSpecificHelper.cpp

```
#include <mlpack/core.hpp>
#include <iostream>
#include <sstream>
//#include </home/tim/Desktop/rlnn4/TrainingDataBuffer.cpp>
#include <cmath>

#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <mlpack/prereqs.hpp>
#include <fstream>
#include <boost/archive/tmpdir.hpp>
#include <boost/serialization/base_object.hpp>
#include <boost/serialization/utility.hpp>
#include <boost/serialization/list.hpp>
#include <boost/serialization/assume_abstract.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/serialization/serialization.hpp>
#include "/home/tim/Desktop/rlnn5/Logging.cpp"

using namespace mlpack;

namespace boost {
namespace serialization {
class access;
}
}

class ApplicationSpecificHelper {
public:
  ApplicationSpecificHelper(const arma::mat &actionList,
                double frameSize,
                double maxEsN0,
                double minEsN0,
                arma::Row<int> modList,
                arma::Row<double> codList,
                arma::Row<double> rollOffList,
                arma::Row<double> symbolRateList,
                arma::Row<double> transmitPowerList,
                arma::Row<int> modCodList,
                double maxBER
                );

  void genNNExploreInputs(arma::mat &inputs);
  void genNNExploitInputs(arma::mat &inputs);
  void processMeasurements(const arma::rowvec &measurementVec);
  void rollBackExploitInputs();
  void updateNNExploitInputs();
  void updateLastNNExploitInputs();
  void forceSetNNExploitInputs(arma::colvec &newExploitInputs);
  bool lastAndNewNNExploitInputEqual();
  bool lastNNExploitInputEmpty();
  void getFitnessParams(arma::rowvec &params);
  void setFitnessObserved(double fObserved, bool exploitFlag);
  void genTrainingSample(arma::colvec &outVec);
  int processNNExploitOutputs(std::vector<arma::mat> &predictedActionVec);
  int getFitObservedOutVecIdx();
  double getRollBackThreshold();
  int returnFallBackAction();
  TrainingDataBuffer::InpOutBuffParams * getNNInpOutBuffParms(int nn);


  void saveCurrentRun(std::string filename) {

    std::ofstream ofs(filename);
    //save data to archvie
    boost::archive::text_oarchive oa(ofs);
    //write class instance to archive
    oa & *this;
    //archive and stream closed with destructors are called
  };

  void loadOldRun(std::string filename) {
    std::ifstream ifs(filename);
    boost::archive::text_iarchive ia(ifs);
    //read class state from archive
    ia & *this;
    //archive and stream closed with destructors are called.
  }

private:
  arma::mat _actionListNormWithNSNR;
  arma::mat _actionList;
  bool _genNNExploreInputsCalledAlready;

  double _measuredEsN0Lin;
  double _measuredPowConsumedLin;
  double _measuredPowEfficiencyLog10;
  double _measuredPowConsumedLinComplement;
```

```
89    double _measuredBandwidth;
90    double _measuredThroughput;
91    double _measuredSpectralEff;
92    double _measuredBEREst;
93    double _measuredBEREstdB;
94
95    double _frameSize;
96    double _maxEsN0Lin;
97    double _minEsN0Lin;
98    arma::Row<int> _modList;
99    arma::Row<double> _codList;
100   arma::Row<double> _rollOffList;
101   arma::Row<double> _symbolRateList;
102   arma::Row<double> _transmitPowerList;
103   arma::Row<int> _modCodList;
104
105   double _RsMax;
106   double _RsMin;
107   double _BWMin;
108   double _BWMax;
109   double _TMax;
110   double _TMin;
111   double _EsMinLin;
112   double _EsMaxLin;
113   double _PConsumMinLin;
114   double _PConsumMaxLin;
115   double _SpectEffMin;
116   double _SpectEffMax;
117   double _PEffMaxLog10;
118   double _PEffMinLog10;
119   double _berDBMax;
120   double _berDBMin;
121
122   std::vector<std::vector<double>> _esnoValuesTable;
123   std::vector<std::vector<double>> _ferValuesTable;
124
125   arma::rowvec _fitObservedParams;
126   double _fitObserved;
127   std::vector<double> _fitObservedBuffer;
128   int _fitObservedBufferPtr;
129
130   arma::colvec _nnExploitInput;
131   arma::colvec _nnExploitInputLast;
132
133   TrainingDataBuffer::InpOutBuffParams _inpOutBuffParamsExplore;
134   TrainingDataBuffer::InpOutBuffParams _inpOutBuffParamsExploit;
135
136   arma::colvec _modClassTargets;
137
138   int _fallBackActionID;
139
140   double estimateBER(double esN0dB, int M, double rate);
141
142   friend class boost::serialization::access;
143   template<class Archive>
144   void serialize(Archive & ar, const int version) {
145     ar & _genNNExploreInputsCalledAlready;
146
147     ar & _measuredEsN0Lin;
148     ar & _measuredPowConsumedLin;
149     ar & _measuredPowConsumedLinComplement;
150     ar & _measuredPowEfficiencyLog10;
151     ar & _measuredBandwidth;
152     ar & _measuredThroughput;
153     ar & _measuredSpectralEff;
154     ar & _measuredBEREst;
155     ar & _measuredBEREstdB;
156
157     ar & _fitObservedParams;
158     ar & _fitObserved;
159     ar & _fitObservedBuffer;
160     ar & _fitObservedBufferPtr;
161
162     ar & _nnExploitInput;
163     ar & _nnExploitInputLast;
164
165     ar & _esnoValuesTable;
166     ar & _ferValuesTable;
167   }
168 };
169
170 ApplicationSpecificHelper::ApplicationSpecificHelper(
171   const arma::mat &actionList, double frameSize, double maxEsN0, double minEsN0,
172   arma::Row<int> modList, arma::Row<double> codList, arma::Row<double> rollOffList,
173   arma::Row<double> symbolRateList, arma::Row<double> transmitPowerList, arma::Row<int>
      modCodList,
174   double maxBER)
175 {
176   //generate normalized action list
177   double actionListMax = actionList.max();
178   double actionListMin = actionList.min();
179   _actionListNormWithNSNR.set_size(actionList.n_rows+1,actionList.n_cols); //last row is SNR
```

```
180   _actionList.set_size(arma::size(actionList));
181   for(int i=0; i<actionList.n_rows; i++) {
182     for(int j=0; j<actionList.n_cols; j++) {
183       _actionListNormWithNSNR(i,j) = (1-(-1))*(actionList(i,j)-actionListMin)/
184             (actionListMax-actionListMin) + (-1);
185       _actionList(i,j) = actionList(i,j);
186     }
187   }
188   for(int j=0; j<actionList.n_cols; j++) {
189     _actionListNormWithNSNR(_actionListNormWithNSNR.n_rows-1,j) = -1; //populate with dummy
      right now
190   }
191
192   //init vars
193   _genNNExploreInputsCalledAlready = false;
194   _fitObservedParams.zeros(7); //row vec [T,BER,W,SpecEff,PEff,PCons,SNRLin]
195   _fitObserved = 0.0;
196   _nnExploitInput.zeros(6,1); //mat (colvec)
197   _nnExploitInputLast = -2*arma::ones(6,1); //mat (colvec)
198
199   //save values
200   _frameSize = frameSize;
201   _modList = modList;
202   _codList = codList;
203   _rollOffList = rollOffList;
204   _symbolRateList = symbolRateList;
205   _transmitPowerList = transmitPowerList;
206   _modCodList = modCodList;
207
208   //compute value mins/maxes
209   _maxEsN0Lin = pow(10,maxEsN0/10);
210   _minEsN0Lin = pow(10,minEsN0/10);
211   _RsMax = symbolRateList.max();
212   _RsMin = symbolRateList.min();
213   //_BWMin = _RsMax*(1+_rollOffList.max());
214   //_BWMax = _RsMin*(1+_rollOffList.min());
215   _BWMin = _RsMin*(1+_rollOffList.min());
216   _BWMax = _RsMax*(1+_rollOffList.max());
217   _TMax = _RsMax*log2(_modList.max())*_codList.max();
218   _TMin = _RsMin*log2(_modList.min())*_codList.min();
219   _EsMinLin = pow(10.0,transmitPowerList.min()/10);
220   _EsMaxLin = pow(10.0,transmitPowerList.max()/10);
221   _PConsumMinLin = _EsMinLin*_RsMin;
222   _PConsumMaxLin = _EsMaxLin*_RsMax;
223   _SpectEffMin = log2(_modList.min())*_codList.min()/(1+_rollOffList.max());
224   _SpectEffMax = log2(_modList.max())*_codList.max()/(1+_rollOffList.min());
225   _PEffMaxLog10 = log10(log2(_modList.max())*_codList.max()/(_EsMinLin*_RsMin));
226   _PEffMinLog10 = log10(log2(_modList.min())*_codList.min()/(_EsMaxLin*_RsMax));
227   _berDBMax = -10*log10(maxBER);
228   _berDBMin = -10*log10(1);
229
230   //nnExplore Buff Params
231   _inpOutBuffParamsExplore.inpBuffIdxs.push_back(-1); //actions
232   _inpOutBuffParamsExplore.inpBuffIdxs.push_back(7); //EsN0Lin
233   _inpOutBuffParamsExplore.inpBuffExpandActions.push_back(2); //actions expanded scaled from row
      max/min
234   _inpOutBuffParamsExplore.inpBuffExpandActions.push_back(0); //EsN0 not expanded
235   _inpOutBuffParamsExplore.inpNormParams << 0 << 1 << -1 << 1 << arma::endr //scale to [-1,1]
236                          << _minEsN0Lin << _maxEsN0Lin << -1 << 1 << arma::endr;
237   _inpOutBuffParamsExplore.outBuffIdxs.push_back(0); //fitObserved
238   _inpOutBuffParamsExplore.outBuffExpandActions.push_back(0); //no expanding
239   _inpOutBuffParamsExplore.outNormParams << 0 << 1 << 0 << 1 << arma::endr; //scale to [0,1] (
      from [0,1])
240
241   //nnExploit Buff Params
242   _inpOutBuffParamsExploit.inpBuffIdxs.push_back(1); //throughput
243   _inpOutBuffParamsExploit.inpBuffIdxs.push_back(2); //BERdB
244   _inpOutBuffParamsExploit.inpBuffIdxs.push_back(3); //BW
245   _inpOutBuffParamsExploit.inpBuffIdxs.push_back(4); //specEff
246   _inpOutBuffParamsExploit.inpBuffIdxs.push_back(5); //PEff
247   _inpOutBuffParamsExploit.inpBuffIdxs.push_back(6); //PConsum
248   _inpOutBuffParamsExploit.inpBuffIdxs.push_back(7); //EsN0Lin
249   _inpOutBuffParamsExploit.inpBuffExpandActions.push_back(0); //not expand
250   _inpOutBuffParamsExploit.inpBuffExpandActions.push_back(0); //not expand
251   _inpOutBuffParamsExploit.inpBuffExpandActions.push_back(0); //not expand
252   _inpOutBuffParamsExploit.inpBuffExpandActions.push_back(0); //not expand
253   _inpOutBuffParamsExploit.inpBuffExpandActions.push_back(0); //not expand
254   _inpOutBuffParamsExploit.inpBuffExpandActions.push_back(0); //not expand
255   _inpOutBuffParamsExploit.inpBuffExpandActions.push_back(0); //not expand
256   _inpOutBuffParamsExploit.inpNormParams  << _TMin << _TMax << -1 << 1 << arma::endr
257                        << _berDBMin << _berDBMax << -1 << 1 << arma::endr
258                        << _BWMin << _BWMax << -1 << 1 << arma::endr
259                        << _SpectEffMin << _SpectEffMax << -1 << 1 << arma::endr
260                        << _PEffMinLog10 << _PEffMaxLog10 << -1 << 1 << arma::endr
261                        << _PConsumMinLin << _PConsumMaxLin << -1 << 1 << arma::endr
262                        << _minEsN0Lin << _maxEsN0Lin << -1 << 1 << arma::endr;
263   _inpOutBuffParamsExploit.outBuffIdxs.push_back(-1); //actions
264   _inpOutBuffParamsExploit.outBuffExpandActions.push_back(2); //expand using row max/min (not
      actionList global)
265   _inpOutBuffParamsExploit.outNormParams << 0 << 0 << 0 << 1 << arma::endr; //since we're using
      action max/min
266                                     //the original min/max doesn't matter.
```

```
267
268   //creating mod class targets
269   arma::Mat<int> uniqueMods = arma::unique(_modList); //col vec
270   _modClassTargets.set_size(uniqueMods.n_elem*uniqueMods.n_elem);
271   int it=0;
272   for(int i=0; i<uniqueMods.n_elem; i++) {
273     for(int j=0; j<uniqueMods.n_elem; j++) {
274       //std::cout << ((double) (uniqueMods(i)-uniqueMods.min()))/((double) (uniqueMods.max()-
      uniqueMods.min())) << "  "
275       //      << (log2((double) uniqueMods(j))-log2((double) uniqueMods.min()))/(log2((double)
      uniqueMods.max())-log2((double) uniqueMods.min())) << std::endl;
276       _modClassTargets(it) = ((double) (uniqueMods(i)-uniqueMods.min()))/((double) (uniqueMods.
      max()-uniqueMods.min()))
277                   + (log2((double) uniqueMods(j))-log2((double) uniqueMods.min()))/(log2((double)
      uniqueMods.max())-log2((double) uniqueMods.min()));
278       it++;
279     }
280   }
281   std::cout << "UNIQUE MODS: " << uniqueMods << std::endl;
282   std::cout << "MODCLASS TARGETS: " << _modClassTargets << std::endl;
283
284   _fitObservedBufferPtr = 0;
285   _fitObservedBuffer.resize(200);
286   for(int i=0; i<_fitObservedBuffer.size(); i++) {
287     _fitObservedBuffer[i] = 0;
288   }
289
290   //load in FER curves
291   std::string line;
292   int lineCount;
293   int lineIter;
294
295   //count number of lines
296   std::ifstream ferCurvesFile("ferCurves.txt");
297   lineCount=0;
298   while(std::getline(ferCurvesFile,line)) {
299     lineCount++;
300   }
301   ferCurvesFile.close();
302   std::cout << "LINE COUNT: "<<lineCount << std::endl;
303
304   //read in values
305   _esnoValuesTable.resize(lineCount);
306   _ferValuesTable.resize(lineCount);
307   lineIter=0;
308   std::ifstream ferCurvesFile2("ferCurves.txt");
309   while(std::getline(ferCurvesFile2,line)) {
310     std::istringstream ss(line);
311     std::string token;
312
313     int i=0;
314     while(std::getline(ss,token,',')) {
315       if((i%2)==0) {
316         _esnoValuesTable[lineIter].push_back(std::atof(token.c_str()));
317       } else {
318         _ferValuesTable[lineIter].push_back(std::atof(token.c_str()));
319       }
320       i++;
321     }
322     lineIter++;
323   }
324   ferCurvesFile2.close();
325
326   // #ifdef LOGGING
327   // debugLogFile << "past ferCurvesFile2.close"<<std::endl;
328   // #endif
329
330   //brute force search for fall back id
331   for(int i=0; i<_actionList.n_cols; i++) {
332     if( _actionList(0,i)==_actionList.row(0).min() &&
333       _actionList(1,i)==_actionList.row(1).max() &&
334       _actionList(3,i)==_actionList.row(3).max() &&
335       _actionList(4,i)==_actionList.row(4).min() &&
336       _actionList(5,i)==_actionList.row(5).min()
337       ) {
338
339       _fallBackActionID = i;
340     }
341
342   }
343
344 }
345
346 void ApplicationSpecificHelper::genNNExploreInputs(arma::mat &inputs) {
347   if(_genNNExploreInputsCalledAlready==false) {
348     inputs.set_size(_actionListNormWithNSNR.n_rows,_actionListNormWithNSNR.n_cols);
349     for(int i=0; i<inputs.n_rows-1; i++) {
350       for(int j=0; j<inputs.n_cols; j++) {
351         inputs(i,j) = _actionListNormWithNSNR(i,j);
352       }
353     }
354     _genNNExploreInputsCalledAlready = true;
```

```
355   }
356   for(int i=0; i<inputs.n_cols; i++) {
357     //inputs(inputs.n_rows-1,i) = ((_measuredEsN0Lin/_maxEsN0Lin)-0.5)/0.5; //scale [-1,1]
358     inputs(inputs.n_rows-1,i) = (1-(-1))*((_measuredEsN0Lin-_minEsN0Lin)/(_maxEsN0Lin-
        _minEsN0Lin)) + (-1);
359   }
360 }
361
362 void ApplicationSpecificHelper::genNNExploitInputs(arma::mat &inputs) {
363   inputs.set_size(_nnExploitInput.n_elem+1,1);
364   for(int i=0; i<_nnExploitInput.n_elem; i++) {
365     inputs(i,0) = (_nnExploitInput(i,0)-0.5)/0.5; //scales [-1,1]
366   }
367   inputs(_nnExploitInput.n_elem) = (1-(-1))*((_measuredEsN0Lin-_minEsN0Lin)/(_maxEsN0Lin-
        _minEsN0Lin)) + (-1);
368 }
369
370 void ApplicationSpecificHelper::updateNNExploitInputs() {
371   for(int i=0; i<_nnExploitInput.n_elem; i++) {
372     _nnExploitInput(i) = _fitObservedParams(i);
373   }
374 }
375
376 void ApplicationSpecificHelper::forceSetNNExploitInputs(arma::colvec &newExploitInputs) {
377   for(int i=0; i<_nnExploitInput.n_elem; i++) {
378     _nnExploitInput(i) = (newExploitInputs(i)+1.0)/2.0; //newExploitInputs is [-1,1], we need to
        convert to [0,1]
379                                         //because that's what genNNExploitInputs expects
380   }
381   //std::cout << "_nnExploitInput now forced to: " << _nnExploitInput << std::endl;
382 }
383
384 void ApplicationSpecificHelper::updateLastNNExploitInputs() {
385   for(int i=0; i<_nnExploitInput.n_elem; i++) {
386     _nnExploitInputLast(i) = _nnExploitInput(i);
387   }
388 }
389
390 bool ApplicationSpecificHelper::lastNNExploitInputEmpty() {
391   bool flag = true;
392   for(int i=0; i<_nnExploitInput.n_elem; i++) {
393     flag = flag && (std::abs(-2.0-_nnExploitInputLast(i))<1e-12);  //are they approx equal
394   }
395   return flag;
396 }
397
398 void ApplicationSpecificHelper::rollBackExploitInputs() {
399   for(int i=0; i<_nnExploitInput.n_elem; i++) {
400     _nnExploitInput(i) = _nnExploitInputLast(i);
401   }
402 }
403
404 bool ApplicationSpecificHelper::lastAndNewNNExploitInputEqual() {
405   bool flag = true;
406   for(int i=0; i<_nnExploitInput.n_elem; i++) {
407     flag = flag && (std::abs(_nnExploitInput(i)-_nnExploitInputLast(i))<1e-12);  //are they
        approx equal
408   }
409   return flag;
410 }
411
412 void ApplicationSpecificHelper::getFitnessParams(arma::rowvec &params) {
413   params.set_size(_fitObservedParams.n_elem-1); //we exclude SNR
414   for(int i=0; i<params.n_elem; i++) {
415     params(i) = _fitObservedParams(i);
416   }
417
418   //reality check
419   if(params(1) == 0.0) { //if the BER=1
420     for(int i=0; i< params.n_elem; i++) {
421       params(i) = 0.0;
422     }
423   }
424 }
425
426 void ApplicationSpecificHelper::setFitnessObserved(double fObserved, bool exploitFlag) {
427   _fitObserved = fObserved;
428
429   if(exploitFlag) {
430     _fitObservedBuffer[_fitObservedBufferPtr] = fObserved;
431     _fitObservedBufferPtr++;
432     if(_fitObservedBufferPtr == _fitObservedBuffer.size()) {
433       _fitObservedBufferPtr = 0;
434     }
435   }
436 }
437
438 double ApplicationSpecificHelper::getRollBackThreshold() {
439   //find max value
440   double thresh = 0.0;
441   for(int i=0; i<_fitObservedBuffer.size(); i++) {
442     if(_fitObservedBuffer[i] > thresh) {
```

```
443         thresh = _fitObservedBuffer[i];
444       }
445     }
446     return 0.9*thresh;
447 }
448
449 int ApplicationSpecificHelper::returnFallBackAction() {
450     std::cout << "falling back" << std::endl;
451     return _fallBackActionID;
452
453 }
454
455
456 void ApplicationSpecificHelper::genTrainingSample(arma::colvec &outVec) {
457     outVec.set_size(8);
458     outVec(0) = _fitObserved;
459     outVec(1) = _measuredThroughput;
460     outVec(2) = _measuredBEREstdB;
461     outVec(3) = _measuredBandwidth;
462     outVec(4) = _measuredSpectralEff;
463     outVec(5) = _measuredPowEfficiencyLog10;
464     outVec(6) = _measuredPowConsumedLinComplement;
465     outVec(7) = _measuredEsN0Lin;
466 }
467
468 int ApplicationSpecificHelper::getFitObservedOutVecIdx() {
469     return 0; //see outVec defn in genTrainingSample
470 }
471
472 TrainingDataBuffer::InpOutBuffParams * ApplicationSpecificHelper::getNNInpOutBuffParms(int nn) {
473     if(nn==0) {
474       return &_inpOutBuffParamsExplore;
475     } else if(nn==1) {
476       return &_inpOutBuffParamsExploit;
477     } else {
478       return NULL;
479     }
480
481 }
482
483 void ApplicationSpecificHelper::processMeasurements(const arma::rowvec &measurementVec) {
484     //measurement vector:
485     //EsN0 (dB), TX Power (dB), Symbol Rate (symbols/sec), Roll Off, Modulation, Code Rate
486
487     double esN0 = measurementVec(0);
488     double esAdd = measurementVec(1);
489     double Rs = measurementVec(2);
490     double roll_off = measurementVec(3);
491     int M = measurementVec(4);
492     double rate = measurementVec(5);
493
494     _measuredEsN0Lin = pow(10.0,(esN0-esAdd)/10.0);
495     _measuredPowConsumedLin = pow(10.0,esAdd/10.0)*Rs;
496     _measuredPowConsumedLinComplement = _PConsumMaxLin+_PConsumMinLin - _measuredPowConsumedLin;
497     _measuredPowEfficiencyLog10 = log10((log2(M)*rate)/_measuredPowConsumedLin);
498     _measuredBandwidth = Rs*(1.0+roll_off);
499     _measuredThroughput = Rs*log2(M)*rate;
500     _measuredSpectralEff = log2(M)*rate/(1.0+roll_off);
501     _measuredBEREst = estimateBER(esN0,M,rate);
502     _measuredBEREstdB = -10.0*log10(_measuredBEREst);
503
504     //populate observed params, normalized to [0,1]
505     _fitObservedParams(0) = (_measuredThroughput-_TMin)/(_TMax-_TMin);
506     _fitObservedParams(1) = (_measuredBEREstdB-_berDBMin)/(_berDBMax-_berDBMin);
507     _fitObservedParams(2) = (_measuredBandwidth-_BWMin)/(_BWMax-_BWMin);
508     _fitObservedParams(3) = (_measuredSpectralEff-_SpectEffMin)/(_SpectEffMax-_SpectEffMin);
509     _fitObservedParams(4) = (_measuredPowEfficiencyLog10-_PEffMinLog10)/(_PEffMaxLog10-
        _PEffMinLog10);
510     _fitObservedParams(5) = (_measuredPowConsumedLinComplement-_PConsumMinLin)/(_PConsumMaxLin-
        _PConsumMinLin);
511     _fitObservedParams(6) = (_measuredEsN0Lin-_minEsN0Lin)/(_maxEsN0Lin-_minEsN0Lin);
512
513 }
514
515 int ApplicationSpecificHelper::processNNExploitOutputs(std::vector<arma::mat> &
        predictedActionVec) {
516     // classify modcod
517     arma::Row<int> uniqueMods = arma::unique(_modList); //col vec
518     arma::rowvec modClassTargetsShort;
519     modClassTargetsShort.zeros(uniqueMods.n_elem);
520     for(int i=0; i<uniqueMods.n_elem*4; i=i+4) {
521       modClassTargetsShort(i/4) = _modClassTargets(i);
522     }
523     int modcodClassIdx = (arma::abs(modClassTargetsShort-(predictedActionVec[2](0)+
        predictedActionVec[4](0)))).index_min();
524
525     //denormalize predicted action
526     double actionPredicted;
527     arma::colvec actionPredictedDiscrete;
528     actionPredictedDiscrete.zeros(_actionList.n_rows);
529     for(int i=0; i<_actionList.n_rows; i++) {
530       double aLMax = _actionList.row(i).max();
```

```
531     double aLMin = _actionList.row(i).min();
532     actionPredicted = (aLMax-aLMin)*(predictedActionVec[i](0)-0)/(1-0) + aLMin;
533     switch(i) {
534       case 0: {
535         actionPredictedDiscrete(i) = _symbolRateList((arma::abs(actionPredicted -
      _symbolRateList)).index_min());
536         break;
537       }
538       case 1: {
539         actionPredictedDiscrete(i) = _transmitPowerList((arma::abs(actionPredicted -
      _transmitPowerList)).index_min());
540         break;
541       }
542       case 2: {
543         arma::rowvec log2Mod;
544         log2Mod.zeros(uniqueMods.n_elem);
545         for(int i=0; i<uniqueMods.n_elem; i++) {
546           log2Mod(i) = log2(uniqueMods(i));
547         }
548         actionPredictedDiscrete(i) = log2Mod(modcodClassIdx);
549         break;
550       }
551       case 3: {
552         actionPredictedDiscrete(i) = _rollOffList((arma::abs(actionPredicted - _rollOffList)).
      index_min());
553         break;
554       }
555       case 4: {
556         actionPredictedDiscrete(i) = uniqueMods(modcodClassIdx);
557         break;
558       }
559       case 5: {
560         arma::rowvec codEdges = arma::trans(_codList(arma::find(_modList==
      actionPredictedDiscrete(4))));
561         actionPredictedDiscrete(i) = codEdges((arma::abs(actionPredicted - codEdges)).index_min
      ());
562         break;
563       }
564     }
565   }
566
567   //find discretized action
568   int actionID = -1;
569   for(int i=0; i<_actionList.n_cols; i++) {
570     if(arma::approx_equal(_actionList.col(i),actionPredictedDiscrete,"reldiff",0.01)) { //find
      within 1% error
571       actionID = i;
572       break;
573     }
574   }
575   return actionID; //if we can't find an action ID, then return -1.
576 }
577
578 double ApplicationSpecificHelper::estimateBER(double EsN0dB, int mod, double code) {
579   double FERlog10;
580   double FERlin;
581
582   /*arma::mat linearPoints = {  {-2.40, 1.10*pow(10,-3), -2.10, 1.10*pow(10,-10)}, //QPSK 1/4
583                 {-1.40, 1.10*pow(10,-3), -1.10, 1.10*pow(10,-10)}, //QPSK 1/3
584                 {-0.40, 1.10*pow(10,-3), -0.10, 1.10*pow(10,-10)}, //QPSK 2/5
585                 { 0.90, 1.10*pow(10,-3),  1.20, 1.10*pow(10,-10)}, //QPSK 1/2
586                 { 2.20, 1.15*pow(10,-3),  2.50, 1.00*pow(10, -8)}, //QPSK 3/5
587                 { 3.35, 1.15*pow(10,-3),  3.70, 1.00*pow(10, -8)}, //QPSK 3/4
588                 { 4.50, 1.10*pow(10,-3),  4.90, 7.00*pow(10,-10)}, //QPSK 4/5
589                 { 5.25, 1.10*pow(10,-3),  5.70, 7.00*pow(10,-10)}, //QPSK 5/6
590                 { 6.00, 5.00*pow(10,-3),  6.50, 5.50*pow(10, -9)}, //QPSK 8/9
591                 { 6.80, 5.00*pow(10,-3),  7.20, 5.50*pow(10, -9)}, //QPSK 9/10 **************
592                 { 5.10, 5.50*pow(10,-4),  5.40, 2.00*pow(10, -8)}, //8PSK 3/5
593                 { 6.50, 5.50*pow(10,-4),  6.80, 2.00*pow(10, -8)}, //8PSK 2/3
594                 { 7.90, 1.15*pow(10,-4),  8.20, 2.50*pow(10, -9)}, //8PSK 3/4
595                 { 8.65, 1.15*pow(10,-4),  8.95, 2.50*pow(10, -9)}, //8PSK 4/5
596                 { 9.40, 7.00*pow(10,-4),  9.70, 5.00*pow(10, -9)}, //8PSK 5/6
597                 {10.70, 7.00*pow(10,-4), 11.00, 1.20*pow(10, -9)}, //8PSK 8/9
598                 {12.00, 7.00*pow(10,-4), 12.30, 1.20*pow(10, -9)}, //8PSK 9/10
599                 { 9.35, 1.00*pow(10,-3),  9.75, 1.00*pow(10, -8)}, //16APSK 2/3
600                 {10.10, 1.00*pow(10,-3), 10.50, 1.00*pow(10, -8)}, //16APSK 3/4
601                 {10.85, 1.00*pow(10,-3), 11.20, 1.00*pow(10, -8)}, //16APSK 4/5
602                 {11.60, 1.00*pow(10,-3), 11.90, 5.00*pow(10, -9)}, //16APSK 5/6
603                 {12.80, 6.00*pow(10,-4), 13.20, 3.00*pow(10, -9)}, //16APSK 8/0
604                 {14.00, 6.00*pow(10,-4), 14.40, 3.00*pow(10, -9)}, //16APSK 9/10
605                 {12.30, 1.00*pow(10,-3), 12.80, 3.00*pow(10, -8)}, //32APSK 3/4
606                 {13.05, 1.00*pow(10,-3), 13.45, 3.00*pow(10, -8)}, //32APSK 4/5
607                 {13.80, 1.00*pow(10,-3), 14.10, 5.00*pow(10, -9)}, //32APSK 5/6
608                 {14.90, 6.00*pow(10,-4), 15.40, 3.00*pow(10, -9)}, //32APSK 8/9
609                 {16.00, 6.00*pow(10,-4), 16.50, 3.00*pow(10, -9)}  //32APSK 9/10
610               };
611
612   arma::uvec modIdxs = arma::find(_modList==mod);
613   int rowIdx;
614   double dist=100.0; //arbitrarily large
615   for(int i=0; i<modIdxs.n_elem; i++) {
616     if(std::abs(_codList(modIdxs(i))-code)<dist) {
```

```
617         dist = std::abs(_codList(modIdxs(i))-code);
618         rowIdx = modIdxs(i);
619       }
620     }
621
622     //linearly interpolate dB curve
623     BERlog10 = (log10(linearPoints(rowIdx,3))-log10(linearPoints(rowIdx,1)))/(linearPoints(rowIdx
        ,2)-linearPoints(rowIdx,0))
624           *(EsN0dB-linearPoints(rowIdx,0))+log10(linearPoints(rowIdx,1));
625     //hard limit at 0 and -12
626     if(BERlog10>=0) {
627       BERlog10=0;
628     }
629     if(BERlog10<-12) {
630       BERlog10 = -12;
631     }
632     //convert to linear
633     BERlin = pow(10,BERlog10);
634     */
635
636     //find modcod
637     arma::uvec modIdxs = arma::find(_modList==mod);
638     int rowIdx;
639     double dist=100.0; //arbitrarily large
640     for(int i=0; i<modIdxs.n_elem; i++) {
641       if(std::abs(_codList(modIdxs(i))-code)<dist) {
642         dist = std::abs(_codList(modIdxs(i))-code);
643         rowIdx = modIdxs(i);
644       }
645     }
646
647     //find closest bounding points on curve
648     int minPoint = -1;
649     int maxPoint = -1;
650     for(int i=0; i<_esnoValuesTable[rowIdx].size(); i++) {
651       if(EsN0dB>=_esnoValuesTable[rowIdx][i]) {
652         minPoint = i;
653       }
654       if(EsN0dB<_esnoValuesTable[rowIdx][i] && maxPoint==-1) {
655         maxPoint = i;
656       }
657     }
658
659     if(minPoint==-1) { //to the left of the fer curve
660       minPoint = maxPoint; //use first two points and linearly interpolate leftward
661       maxPoint++;
662     }
663     if(maxPoint==-1) { //to the right of the fer cruve
664       maxPoint = minPoint; //use last two points and linearly interpolate rightward
665       minPoint--;
666     }
667
668     FERlog10 = ((log10(_ferValuesTable[rowIdx][maxPoint])-log10(_ferValuesTable[rowIdx][minPoint])
        )/(_esnoValuesTable[rowIdx][maxPoint]-_esnoValuesTable[rowIdx][minPoint]))
669           *(EsN0dB-_esnoValuesTable[rowIdx][minPoint])+log10(_ferValuesTable[rowIdx][minPoint]);
670
671     //hard limit at 0 and -12
672     if(FERlog10>=0) {
673       FERlog10=0;
674     }
675     if(FERlog10<-6) {
676       FERlog10 = -6;
677     }
678     //convert to linear
679     FERlin = pow(10,FERlog10);
680
681     return FERlin;
682   }
```

D.8 ThreeLayerNetwork.cpp

```
#ifndef THREELAYERNETWORK
#define THREELAYERNETWORK

#include <mlpack/core.hpp>

#include <mlpack/methods/ann/activation_functions/logistic_function.hpp>
#include <mlpack/methods/ann/activation_functions/identity_function.hpp>

#include <mlpack/methods/ann/init_rules/random_init.hpp>

#include <mlpack/methods/ann/layer/linear_layer.hpp>
#include <mlpack/methods/ann/layer/base_layer.hpp>
#include <mlpack/methods/ann/layer/identity_output_layer.hpp>

#include <mlpack/methods/ann/ffn.hpp>
#include <mlpack/methods/ann/performance_functions/mse_function.hpp>
#include <mlpack/core/optimizers/rmsprop/rmsprop.hpp>

#include "/home/tim/Desktop/rlnn5/Logging.hpp"

#include <iostream>

using namespace mlpack;


class ThreeLayerNetwork {
  public:

  std::tuple <  ann::LinearLayer<>,
          ann::BaseLayer<ann::LogisticFunction>,
          ann::LinearLayer<>,
          ann::BaseLayer<ann::LogisticFunction>,
          ann::LinearLayer<>,
          ann::BaseLayer<ann::IdentityFunction>
        > modules;

  ann::FFN <  decltype(modules),
        ann::IdentityOutputLayer,
        ann::RandomInitialization,
        ann::MeanSquaredErrorFunction
      > net;

  ThreeLayerNetwork(int inputVectorSize, const std::vector<int> &hiddenLayerSize, int
    outputVectorSize)
      :
        modules(ann::LinearLayer<>(inputVectorSize,hiddenLayerSize[0]),
          ann::BaseLayer<ann::LogisticFunction>(),
          ann::LinearLayer<>(hiddenLayerSize[0], hiddenLayerSize[1]),
          ann::BaseLayer<ann::LogisticFunction>(),
          ann::LinearLayer<>(hiddenLayerSize[1], outputVectorSize),
          ann::BaseLayer<ann::IdentityFunction>()
           ),
        net(modules,ann::IdentityOutputLayer(),ann::RandomInitialization(),ann::
    MeanSquaredErrorFunction())
      {} //constructor

  private:
};
#endif
```

D.9 TwoLayerNetwork.cpp

```
#ifndef TWOLAYERNETWORK
#define TWOLAYERNETWORK

#include <mlpack/core.hpp>

#include <mlpack/methods/ann/activation_functions/logistic_function.hpp>
#include <mlpack/methods/ann/activation_functions/identity_function.hpp>

#include <mlpack/methods/ann/init_rules/random_init.hpp>

#include <mlpack/methods/ann/layer/linear_layer.hpp>
#include <mlpack/methods/ann/layer/base_layer.hpp>
#include <mlpack/methods/ann/layer/identity_output_layer.hpp>

#include <mlpack/methods/ann/ffn.hpp>
#include <mlpack/methods/ann/performance_functions/mse_function.hpp>
#include <mlpack/core/optimizers/rmsprop/rmsprop.hpp>

#include "/home/tim/Desktop/rlnn5/Logging.hpp"

#include <iostream>

using namespace mlpack;


class TwoLayerNetwork {
  public:

  std::tuple <  ann::LinearLayer<>,
          ann::BaseLayer<ann::LogisticFunction>,
          ann::LinearLayer<>,
          ann::BaseLayer<ann::IdentityFunction>
        > modules;

  ann::FFN <  decltype(modules),
        ann::IdentityOutputLayer,
        ann::RandomInitialization,
        ann::MeanSquaredErrorFunction
      > net;

  TwoLayerNetwork(int inputVectorSize, const std::vector<int> &hiddenLayerSize, int
     outputVectorSize)
      :
        modules(ann::LinearLayer<>(inputVectorSize,hiddenLayerSize[0]),
          ann::BaseLayer<ann::LogisticFunction>(),
          ann::LinearLayer<>(hiddenLayerSize[0], outputVectorSize),
          ann::BaseLayer<ann::IdentityFunction>()
           ),
        net(modules,ann::IdentityOutputLayer(),ann::RandomInitialization(),ann::
     MeanSquaredErrorFunction())
      {} //constructor

  private:
};

#endif
```

D.10 identity_output_layer.hpp

```
/**
 * @file linear_layer.hpp
 * @author Marcus Edel
 *
 * Definition of the IdentityOutputLayer class also known as fully-connected layer or
 * affine transformation.
 *
 * mlpack is free software; you may redistribute it and/or modify it under the
 * terms of the 3-clause BSD license.  You should have received a copy of the
 * 3-clause BSD license along with mlpack.  If not, see
 * http://www.opensource.org/licenses/BSD-3-Clause for more information.
 */
#ifndef MLPACK_METHODS_ANN_LAYER_ID_OUTPUT_LAYER_HPP
#define MLPACK_METHODS_ANN_LAYER_ID_OUTPUT_LAYER_HPP

#include <mlpack/core.hpp>
#include <mlpack/methods/ann/layer/layer_traits.hpp>

namespace mlpack {
namespace ann /** Artificial Neural Network. */ {

/**
 * Implementation of the IdentityOutputLayer class. The IdentityOutputLayer class represents a
 * single layer of a neural network.
 *
 * @tparam InputDataType Type of the input data (arma::colvec, arma::mat,
 *         arma::sp_mat or arma::cube).
 * @tparam OutputDataType Type of the output data (arma::colvec, arma::mat,
 *         arma::sp_mat or arma::cube).
 */
template <
    typename InputDataType = arma::mat,
    typename OutputDataType = arma::mat
>
class IdentityOutputLayer
{
 public:
  /**
   * Create the IdentityOutputLayer object using the specified number of units.
   *
   * @param inSize The number of input units.
   * @param outSize The number of output units.
   */
  IdentityOutputLayer()
  {
   }

  /**
   * Ordinary feed forward pass of a neural network, evaluating the function
   * f(x) by propagating the activity forward through f.
   *
   * @param input Input data used for evaluating the specified function.
   * @param output Resulting output activation.
   */
  template<typename eT>
  void Forward(const arma::Mat<eT>& input, arma::Mat<eT>& output)
  {
    output = input;
  }

  /**
   * Ordinary feed forward pass of a neural network, evaluating the function
   * f(x) by propagating the activity forward through f.
   *
   * @param input Input data used for evaluating the specified function.
   * @param output Resulting output activation.
   */
  template<typename eT>
  void Forward(const arma::Cube<eT>& input, arma::Mat<eT>& output)
  {
    arma::Mat<eT> data(input.n_elem, 1);

    for (size_t s = 0, c = 0; s < input.n_slices / data.n_cols; s++)
    {
      for (size_t i = 0; i < data.n_cols; i++, c++)
      {
        data.col(i).subvec(s * input.n_rows * input.n_cols, (s + 1) *
            input.n_rows * input.n_cols - 1) = arma::trans(arma::vectorise(
            input.slice(c), 1));
      }
    }

    output = data;
  }

  /**
   * Ordinary feed backward pass of a neural network, calculating the function
   * f(x) by propagating x backwards trough f. Using the results from the feed
```

```
89      * forward pass.
90      *
91      * @param input The propagated input activation.
92      * @param gy The backpropagated error.
93      * @param g The calculated gradient.
94      */
95     template<typename InputType, typename eT>
96     void Backward(const InputType& /* unused */,
97                   const arma::Mat<eT>& gy,
98                   arma::Mat<eT>& g)
99     {
100      g = weights.t() * gy;
101    }
102
103    /*
104     * Calculate the gradient using the output delta and the input activation.
105     *
106     * @param input The propagated input.
107     * @param error The calculated error.
108     * @param gradient The calculated gradient.
109     */
110    template<typename InputType, typename ErrorType, typename GradientType>
111    void Gradient(const InputType& input,
112                  const ErrorType& error,
113                  GradientType& gradient)
114    {
115      GradientDelta(input, error, gradient);
116    }
117 /////! TMP
118  void OutputClass(InputDataType& input, OutputDataType& output)
119   {
120     output = input;
121
122     }
123    //! Get the weights.
124    OutputDataType const& Weights() const { return weights; }
125    //! Modify the weights.
126    OutputDataType& Weights() { return weights; }
127
128    //! Get the input parameter.
129    InputDataType const& InputParameter() const { return inputParameter; }
130    //! Modify the input parameter.
131    InputDataType& InputParameter() { return inputParameter; }
132
133    //! Get the output parameter.
134    OutputDataType const& OutputParameter() const { return outputParameter; }
135    //! Modify the output parameter.
136    OutputDataType& OutputParameter() { return outputParameter; }
137
138    //! Get the delta.
139    OutputDataType const& Delta() const { return delta; }
140    //! Modify the delta.
141    OutputDataType& Delta() { return delta; }
142
143    //! Get the gradient.
144    OutputDataType const& Gradient() const { return gradient; }
145    //! Modify the gradient.
146    OutputDataType& Gradient() { return gradient; }
147
148    /**
149     * Serialize the layer
150     */
151    template<typename Archive>
152    void Serialize(Archive& ar, const unsigned int /* version */)
153    {
154      ar & data::CreateNVP(weights, "weights");
155    }
156
157   private:
158    /*
159     * Calculate the gradient using the output delta (3rd order tensor) and the
160     * input activation (3rd order tensor).
161     *
162     * @param input The input parameter used for calculating the gradient.
163     * @param d The output delta.
164     * @param g The calculated gradient.
165     */
166    template<typename eT>
167    void GradientDelta(const arma::Cube<eT>& input,
168                       const arma::Mat<eT>& d,
169                       arma::Cube<eT>& g)
170    {
171      g = arma::Cube<eT>(weights.n_rows, weights.n_cols, 1);
172      arma::Mat<eT> data = arma::Mat<eT>(d.n_cols,
173          input.n_elem / d.n_cols);
174
175      for (size_t s = 0, c = 0; s < input.n_slices /
176          data.n_rows; s++)
177      {
178        for (size_t i = 0; i < data.n_rows; i++, c++)
179        {
180          data.row(i).subvec(s * input.n_rows *
```

```
            input.n_cols, (s + 1) *
            input.n_rows *
        input.n_cols - 1) = arma::vectorise(
                input.slice(c), 1);
      }
    }

    g.slice(0) = d * data / d.n_cols;
  }

  /*
   * Calculate the gradient (3rd order tensor) using the output delta
   * (dense matrix) and the input activation (dense matrix).
   *
   * @param input The input parameter used for calculating the gradient.
   * @param d The output delta.
   * @param g The calculated gradient.
   */
  template<typename eT>
  void GradientDelta(const arma::Mat<eT>& input,
                     const arma::Mat<eT>& d,
                     arma::Cube<eT>& g)
  {
    g = arma::Cube<eT>(weights.n_rows, weights.n_cols, 1);
    Gradient(input, d, g.slice(0));
  }

  /*
   * Calculate the gradient (dense matrix) using the output delta
   * (dense matrix) and the input activation (3rd order tensor).
   *
   * @param input The input parameter used for calculating the gradient.
   * @param d The output delta.
   * @param g The calculated gradient.
   */
  template<typename eT>
  void GradientDelta(const arma::Cube<eT>& input,
                     const arma::Mat<eT>& d,
                     arma::Mat<eT>& g)
  {
    arma::Cube<eT> grad = arma::Cube<eT>(weights.n_rows, weights.n_cols, 1);
    Gradient(input, d, grad);
    g = grad.slice(0);
  }

  /*
   * Calculate the gradient (dense matrix) using the output delta
   * (dense matrix) and the input activation (dense matrix).
   *
   * @param input The input parameter used for calculating the gradient.
   * @param d The output delta.
   * @param g The calculated gradient.
   */
  template<typename eT>
  void GradientDelta(const arma::Mat<eT>& input,
                     const arma::Mat<eT>& d,
                     arma::Mat<eT>& g)
  {
    g = d * input.t();
  }

  //! Locally-stored number of input units.
  size_t inSize;

  //! Locally-stored number of output units.
  size_t outSize;

  //! Locally-stored weight object.
  OutputDataType weights;

  //! Locally-stored delta object.
  OutputDataType delta;

  //! Locally-stored gradient object.
  OutputDataType gradient;

  //! Locally-stored input parameter object.
  InputDataType inputParameter;

  //! Locally-stored output parameter object.
  OutputDataType outputParameter;
}; // class IdentityOutputLayer


//! Layer traits for the linear layer.
template<
    typename InputDataType,
    typename OutputDataType
>
class LayerTraits<IdentityOutputLayer<InputDataType, OutputDataType> >
{
 public:
```

```
273   static const bool IsBinary = false;
274   static const bool IsOutputLayer = true;
275   static const bool IsBiasLayer = false;
276   static const bool IsLSTMLayer = false;
277   static const bool IsConnection = true;
278 };
279
280 } // namespace ann
281 } // namespace mlpack
282
283 #endif
```

D.11 MemoryManagement.hpp

```
1  #ifndef _MEM_MANAGE_H_
2  #define _MEM_MANAGE_H_
3
4    #ifndef MAKEFILE_MEM
5    # warning "Falling back on MemoryManagement.hpp"
6    #define LM  true
7    #define RLM false
8    #define NSE false
9
10   #endif
11 #endif
```

D.12 RecursiveLMHelper.cpp

```
#ifndef RECURSIVELMHELPER
#define RECURSIVELMHELPER

#include <mlpack/core.hpp>
#include <iostream>
#include <sstream>
//#include </home/tim/Desktop/rlnn4/TrainingDataBuffer.cpp>
#include <cmath>

#include <mlpack/prereqs.hpp>
#include <fstream>
#include "/home/tim/Desktop/rlnn5/Logging.hpp"


using namespace mlpack;

namespace boost {
namespace serialization {
class access;
}
}

struct Neuron {
  std::vector<double> inputs;
  std::vector<double> weights;
  double netVal;
  double yVal;
  std::string activationType; //"logsig", "linear" supported

  double actFnSlope;
  std::vector<double> deltaVal;
};

class RecursiveLMHelper {
public:


  arma::mat pMat;
  RecursiveLMHelper();
  RecursiveLMHelper(std::vector< std::vector<Neuron> > &network);
  void recursiveUpdate(std::vector<std::vector<Neuron>> &network,
              double mu,
              arma::colvec &errorVec,
              arma::colvec &grad,
              arma::colvec &allCurrentWeights,
              const arma::mat &inpPattern,
              const arma::mat outPatternTrain
              ); //* GOTTA DO THIS TO UPDATE ALL SAMPLES*//

private:

  arma::mat sMat;
  float alpha;
  int timeCount;

};

RecursiveLMHelper::RecursiveLMHelper()
{
  //int networkSize = network.size();
  //pMat = arma::mat(networkSize,networkSize,arma::fill::eye);
  sMat = arma::mat(2,2,arma::fill::eye);
  alpha = 0.98;
  timeCount = 0;
}

RecursiveLMHelper::RecursiveLMHelper(std::vector< std::vector<Neuron> > &network)
{
  int networkSize = network.size();
  pMat = arma::mat(networkSize,networkSize,arma::fill::eye);
  sMat = arma::mat(2,2,arma::fill::eye);
  alpha = 0.98;
  timeCount = 0;
}


void RecursiveLMHelper::recursiveUpdate(std::vector<std::vector<Neuron>> &network,
              double mu,
              arma::colvec &errorVec,
              arma::colvec &grad,
              arma::colvec &allCurrentWeights,
              const arma::mat &inpPattern,
              const arma::mat outPatternTrain
              )
{
  // #ifdef LOGGING
  //  logFile <<"Recursive Update Flag 1" <<std::endl;
  // #endif
```

```
89
90    arma::mat Omega;
91    arma::colvec omegaRow;
92    arma::mat lambdaMat = arma::mat(2,2,arma::fill::eye);
93    const int nOutputs = network[network.size()-1].size();
94    //int networkSize = inpPattern.n_cols * nOutputs;
95    int networkSize = grad.size();
96
97     // #ifdef LOGGING
98     //    logFile <<"Recursive Update Flag 2b,grad: "<<grad.size() <<std::endl;
99     // #endif
100
101   omegaRow.set_size(arma::size(grad));
102
103   // #ifdef LOGGING
104   //  logFile <<"Recursive Update Flag 2,netSize: "<<networkSize <<std::endl;
105   //  logFile <<"inpPattern:"<<inpPattern.n_rows<<","<<inpPattern.n_cols<<std::endl;
106   // #endif
107
108   if (timeCount == 0){
109     pMat = arma::mat(networkSize,networkSize);
110
111   }
112
113   //  int jMatRows=nOutputs*inpPattern.n_cols;
114   //jMat.set_size(jMatRows,jMatCols);
115   //resize error vector
116   //errorVec.set_size(jMatRows);
117   //resize current weight vector
118   //allCurrentWeights.set_size(jMatCols);
119   ////////////////////////////////////////
120   // #ifdef LOGGING
121   //  logFile <<"Recursive Update Flag 3" <<std::endl;
122   // #endif
123
124   omegaRow.zeros();
125   // #ifdef LOGGING
126   //  logFile <<"Recursive Update Flag 3a: "<<timeCount%networkSize <<std::endl;
127   //  logFile <<"omegaRow size: "<< omegaRow.size()<<std::endl;
128   // #endif
129   omegaRow(timeCount % networkSize) = 1;
130   // #ifdef LOGGING
131   //  logFile <<"Recursive Update Flag 3b" <<std::endl;
132   // #endif
133   Omega = arma::join_rows(grad, omegaRow);
134
135   lambdaMat(1,1) = 1/mu;
136   // #ifdef LOGGING
137   //  logFile <<"Recursive Update Flag 4, lambdaMat"<<lambdaMat.n_rows<<","<<lambdaMat.n_cols<<"
        Omega: "<<Omega.n_rows<<","<<Omega.n_cols<< " pMat: "<<pMat.n_rows<<","<<pMat.n_cols <<std
        ::endl;
138   // #endif
139
140   sMat = alpha * lambdaMat + Omega.t() * pMat * Omega;
141   // #ifdef LOGGING
142   //  logFile <<"Recursive Update Flag 4a" <<std::endl;
143   //  sMat.print(logFile);
144   // #endif
145   pMat = 1/alpha * (pMat - pMat * Omega * arma::pinv(sMat) * Omega.t() * pMat);
146   timeCount += 1;
147   // #ifdef LOGGING
148   //  logFile <<"Recursive Update Flag 5" <<std::endl;
149   // #endif
150 }
151
152 #endif
```

D.13 Logging.hpp

```
1  //#ifndef _LOGGING_H_
2  //#define _LOGGING_H_
3
4  #include <fstream>
5
6  #define LOGGING
7
8  extern std::ofstream logFile;
9  // extern std::ofstream debugLogFile;
10 //#endif
```

D.14 Logging.cpp

```
1  #ifndef _LOGGING_CPP_
2  #define _LOGGING_CPP_
3
4  #include <fstream>
5  #include "/home/tim/Desktop/rlnn5/Logging.hpp"
6
7
8  std::ofstream logFile("logging.txt",std::ofstream::out);
9  // std::ofstream debugLogFile("Debug_log.txt",std::ofstream::out);
10
11 #endif
```

D.15 Makefile

```
CPPFLAGS += -std=gnu++11
CPPFLAGS += -larmadillo
CPPFLAGS += -lmlpack
CPPFLAGS += -lboost_serialization
CPPFLAGS += -lboost_system
CPPFLAGS += -lpthread
CPPFLAGS += -lcurl
CPPFLAGS += -g


lm :  CPPFLAGS += -DMAKEFILE_MEM
lm :  CPPFLAGS += -DLM=true
lm :  CPPFLAGS += -DRLM=false
lm :  CPPFLAGS += -DNSE=false

rlm : CPPFLAGS += -DMAKEFILE_MEM
rlm : CPPFLAGS += -DLM=false
rlm : CPPFLAGS += -DRLM=true
rlm : CPPFLAGS += -DNSE=false

nse : CPPFLAGS += -DMAKEFILE_MEM
nse : CPPFLAGS += -DLM=false
nse : CPPFLAGS += -DRLM=false
nse : CPPFLAGS += -DNSE=true

default : RLNNCognitiveEngineTester_v3.cpp
  g++ -o RLNNCognitiveEngineTester_v3 RLNNCognitiveEngineTester_v3.cpp $(CPPFLAGS) 2> err.txt

v2 : RLNNCognitiveEngineTester_v2.cpp
  g++ -o RLNNCognitiveEngineTester_v2 RLNNCognitiveEngineTester_v2.cpp -std=gnu++11  -larmadillo
      -lmlpack -lboost_serialization -lboost_system -pthread -lcurl -O2

all: lm rlm nse

lm : RLNNCognitiveEngineTester_v3.cpp
  g++ -o RLNNCognitiveEngineTester_v3_lm RLNNCognitiveEngineTester_v3.cpp $(CPPFLAGS) 2> err.txt
rlm : RLNNCognitiveEngineTester_v3.cpp
  g++ -o RLNNCognitiveEngineTester_v3_rlm RLNNCognitiveEngineTester_v3.cpp $(CPPFLAGS) 2> err.
     txt
nse : RLNNCognitiveEngineTester_v3.cpp
  g++ -o RLNNCognitiveEngineTester_v3_nse RLNNCognitiveEngineTester_v3.cpp $(CPPFLAGS) 2> err.
     txt


clean:
  rm RLNNCognitiveEngineTester_v3
  rm RLNNCognitiveEngineTester_v3_lm
  rm RLNNCognitiveEngineTester_v3_rlm
  rm RLNNCognitiveEngineTester_v3_nse
```

Appendix E

Pass Details, 2017

Table E.1: Various details about the flight test SNR conditions for the tests conducted in 2017, week 1.

Date	DOY	Start Time (GMT)	End Time (GMT)	Longest period (s) above 63 (C/No)	Longest period (s) above 70 (C/No)	Longest period (s) above 75 (C/No)	Pass Quality	Training Method	Mission
5/2/2017	122	17:13:31	17:21:13	330	270	109	excellent/ great	LM	Emergency
5/2/2017	122	18:50:54	18:56:01	86	0	0	ok/poor	LM	Emergency
5/3/2017	123	11:29:50	11:37:24	307	211	101	excellent/ great	LM	Power Saving
5/3/2017	123	16:21:36	16:29:05	302	198	65	good	LM	Power Saving
5/3/2017	123	17:58:19	18:05:04	237	92	0	good	LM	Emergency
5/4/2017	124	15:29:42	15:36:45	199	40	0	good	LM	Power Saving
5/4/2017	124	17:06:06	17:13:35	323	213	57	excellent/ great	LM	Emergency
5/5/2017	125	14:37:47	14:44:18	61	0	0	ok/poor	LM	Power Saving
5/5/2017	125	16:14:02	16:21:45	377	254	181	excellent/ great	LM	Power Saving

Table E.2: Various details about the flight test SNR conditions for the tests conducted in 2017, week 2.

Date	DOY	Start Time (GMT)	End Time (GMT)	Longest period (s) above 63 (C/No)	Longest period (s) above 70 (C/No)	Longest period (s) above 75 (C/No)	Pass Quality	Training Method	Mission
5/8/2017	128	15:14:24	15:22:04	308	225	122	excellent/ great	LM	Emergency
5/9/2017	129	14:22:21	14:30:01	326	214	144	excellent/ great	LM	Power Saving
5/9/2017	129	15:59:32	16:05:09	146	0	0	good	LM	Cooperation
5/10/2017	130	13:30:22	13:37:45	223	122	34	excellent/ great	LM	Cooperation
5/10/2017	130	15:06:59	15:13:57	311	123	0	good	LM	Power Saving
5/11/2017	131	12:38:23	12:45:18	186	19	0	good	LM	Cooperation
5/11/2017	131	14:14:43	14:22:17	318	218	75	excellent/ great	LM	Cooperation
5/12/2017	132	13:22:35	13:30:18	375	263	192	excellent/ great	LM	Cooperation
5/12/2017	132	15:00:08	15:04:52	78	0	0	ok/poor	LM	Cooperation

Appendix F

Pass Details, 2018

Table F.1: Various details about the flight test SNR conditions for the tests conducted in 2018.

Date	DOY	Start Time (GMT)	End Time (GMT)	Longest period (s) above 63 (C/No)	Longest period (s) above 70 (C/No)	Longest period (s) above 75 (C/No)	Pass Quality	Training Method	Mission
8/16/2018	228	17:25:43	17:31:28	300	61	0	ok/poor	NSE	Emergency
8/16/2018	228	19:03:39	19:08:11	78	27	0	ok/poor	NSE	Power Saving
8/16/2018	228	20:40:14	20:46:33	358	192	30	good	NSE	Emergency
8/16/2018	228	22:16:36	22:23:32	352	321	216	excellent/ great	CE-NSE	Power Saving
8/17/2018	229	14:56:29	15:03:20	416	316	207	excellent/ great	RLM	Emergency
8/17/2018	229	16:33:19	16:39:40	363	200	71	excellent/ great	RLM	Powersaving
8/17/2018	229	19:48:21	19:54:02	298	59	0	ok/poor	RLM	Emergency
8/17/2018	229	21:24:36	21:31:44	405	374	248	excellent/ great	RLM	Cooperation
8/21/2018	233	13:05:20	13:11:47	301	172	0	good	RLM	Emergency
8/21/2018	233	14:41:41	14:48:21	222	119	31	good	RLM	Cooperation
8/21/2018	233	16:19:44	16:24:28	29	0	0	ok/poor	RLM	Cooperation
8/21/2018	233	17:56:59	18:02:16	46	0	0	ok/poor	NSE	Cooperation
8/22/2018	234	18:41:19	18:48:01	220	114	32	good	NSE	Power Saving
8/22/2018	234	20:17:54	20:24:19	311	185	0	good	RLM	Power Saving
8/24/2018	236	12:05:42	12:12:27	318	196	36	good	NSE	Emergency
8/24/2018	236	13:42:22	13:48:51	210	87	4	good	NSE	Cooperation
8/24/2018	236	18:33:45	18:40:51	375	272	198	excellent/ great	NSE	Cooperation